21世纪高等学校规划教材 | 计算机科学与技术

教育部“国家精品在线开放课程”配套教材





# C++语言程序设计
## （MOOC版）（第2版）

阚道宏　编著

清华大学出版社

21 世纪高等学校规划教材·计算机科学与技术

# C++语言程序设计(MOOC 版)

## (第 2 版)

阚道宏　编著

清华大学出版社
北　京

## 内 容 简 介

本书是在《C++语言程序设计(MOOC 版)》的基础上进一步总结爱课程网“中国大学 MOOC”(http://www.icourse163.org/)的网络教学实践修订而成。本书按照实际编程应用来梳理和组织 C++语言的知识点，按章节顺序可分为程序设计基础、结构化程序设计方法和面向对象程序设计方法三大部分。内容编排由易到难，循序渐进。每个小节都设计了适合在线评判的单选练习题，每章则设计了适合课堂讨论的程序阅读题、改错题和编程题。

凡开设“C++语言程序设计”课程的教师可将本书作为授课教材使用，联系作者可免费获得配套教学课件。参加慕课(MOOC)或其他网络课程学习的学生可将本书作为线下阅读教材使用。

**本书封面贴有清华大学出版社防伪标签，无标签者不得销售。**
**版权所有，侵权必究。侵权举报电话：010-62782989　13701121933**

**图书在版编目(CIP)数据**

C++语言程序设计：MOOC 版/阚道宏编著. —2 版. —北京：清华大学出版社，2017（2018.1 重印）
（21 世纪高等学校规划教材・计算机科学与技术）
ISBN 978-7-302-47562-0

Ⅰ. ①C…　Ⅱ. ①阚…　Ⅲ. ①C 语言-程序设计-高等学校-教材　Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2017）第 140800 号

**责任编辑**：郑寅堃
**封面设计**：傅瑞学
**责任校对**：梁　毅
**责任印制**：李红英

**出版发行**：清华大学出版社
**网　　址**：http：//www.tup.com.cn，　http：//www.wqbook.com
**地　　址**：北京清华大学学研大厦 A 座　　**邮　　编**：100084
**社 总 机**：010-62770175　　**邮　　购**：010-62786544
**投稿与读者服务**：010-62776969, c-service@tup.tsinghua.edu.cn
**质量反馈**：010-62772015, zhiliang@tup.tsinghua.edu.cn
**课件下载**：　http：//www.tup.com.cn,010-62795954
**印 装 者**：清华大学印刷厂
**经　　销**：全国新华书店
**开　　本**：185mm×260mm　　**印　张**：29.75　　**字　数**：721 千字
**版　　次**：2016 年 2 月第 1 版　2017 年 10 月第 2 版　　**印　次**：2018 年 1 月第 2 次印刷
**印　　数**：2001～3000
**定　　价**：59.00 元

---

产品编号：074877-02

# 出版说明

随着我国改革开放的进一步深化，高等教育也得到了快速发展，各地高校紧密结合地方经济建设发展需要，科学运用市场调节机制，加大了使用信息科学等现代科学技术提升、改造传统学科专业的投入力度，通过教育改革合理调整和配置了教育资源，优化了传统学科专业，积极为地方经济建设输送人才，为我国经济社会的快速、健康和可持续发展以及高等教育自身的改革发展做出了巨大贡献。但是，高等教育质量还需要进一步提高以适应经济社会发展的需要，不少高校的专业设置和结构不尽合理，教师队伍整体素质亟待提高，人才培养模式、教学内容和方法需要进一步转变，学生的实践能力和创新精神亟待加强。

教育部一直十分重视高等教育质量工作。2007 年 1 月，教育部下发了《关于实施高等学校本科教学质量与教学改革工程的意见》，计划实施“高等学校本科教学质量与教学改革工程（简称‘质量工程’）”，通过专业结构调整、课程教材建设、实践教学改革、教学团队建设等多项内容，进一步深化高等学校教学改革，提高人才培养的能力和水平，更好地满足经济社会发展对高素质人才的需要。在贯彻和落实教育部“质量工程”的过程中，各地高校发挥师资力量强、办学经验丰富、教学资源充裕等优势，对其特色专业及特色课程（群）加以规划、整理和总结，更新教学内容、改革课程体系，建设了一大批内容新、体系新、方法新、手段新的特色课程。在此基础上，经教育部相关教学指导委员会专家的指导和建议，清华大学出版社在多个领域精选各高校的特色课程，分别规划出版系列教材，以配合“质量工程”的实施，满足各高校教学质量和教学改革的需要。

为了深入贯彻落实教育部《关于加强高等学校本科教学工作，提高教学质量的若干意见》精神，紧密配合教育部已经启动的“高等学校教学质量与教学改革工程精品课程建设工作”，在有关专家、教授的倡议和有关部门的大力支持下，我们组织并成立了“清华大学出版社教材编审委员会”（以下简称“编委会”），旨在配合教育部制定精品课程教材的出版规划，讨论并实施精品课程教材的编写与出版工作。“编委会”成员皆来自全国各类高等学校教学与科研第一线的骨干教师，其中许多教师为各校相关院、系主管教学的院长或系主任。

按照教育部的要求，“编委会”一致认为，精品课程的建设工作从开始就要坚持高标准、严要求，处于一个比较高的起点上；精品课程教材应该能够反映各高校教学改革与课程建设的需要，要有特色风格、有创新性（新体系、新内容、新手段、新思路，教材的内容体系有较高的科学创新、技术创新和理念创新的含量）、先进性（对原有的学科体系有实质性的改革和发展，顺应并符合 21 世纪教学发展的规律，代表并引领课程发展的趋势和方向）、示范性（教材所体现的课程体系具有较广泛的辐射性和示范性）和一定的前瞻性。教材由个人申报或各校推荐（通过所在高校的“编委会”成员推荐），经“编委会”认真评审，最

后由清华大学出版社审定出版。目前，针对计算机类和电子信息类相关专业成立了两个“编委会”，即“清华大学出版社计算机教材编审委员会”和“清华大学出版社电子信息教材编审委员会”。推出的特色精品教材包括：

（1）21 世纪高等学校规划教材・计算机应用——高等学校各类专业，特别是非计算机专业的计算机应用类教材。

（2）21 世纪高等学校规划教材・计算机科学与技术——高等学校计算机相关专业的教材。

（3）21 世纪高等学校规划教材・电子信息——高等学校电子信息相关专业的教材。

（4）21 世纪高等学校规划教材・软件工程——高等学校软件工程相关专业的教材。

（5）21 世纪高等学校规划教材・信息管理与信息系统。

（6）21 世纪高等学校规划教材・财经管理与应用。

（7）21 世纪高等学校规划教材・电子商务。

（8）21 世纪高等学校规划教材・物联网。

清华大学出版社经过三十多年的努力，在教材尤其是计算机和电子信息类专业教材出版方面树立了权威品牌，为我国的高等教育事业做出了重要贡献。清华版教材形成了技术准确、内容严谨的独特风格，这种风格将延续并反映在特色精品教材的建设中。

清华大学出版社教材编审委员会
联系人： 魏江江
E-mail:weijj@tup.tsinghua.edu.cn




# 第2版前言

本书是在《C++语言程序设计( MOOC 版)》的基础上进一步总结爱课程网“中国大学MOOC”(http://www.icourse163.org/)的网络教学实践修订而成。依据本教材开设的“C++语言程序设计”MOOC(慕课)课程已开设四个学期，累计选课人数超过十万人。

MOOC 学员能够积极参与课堂讨论，提出各种问题，并对如何开展 C++语言程序设计教学提出了很多宝贵的意见和建议。这里分享几个 MOOC 课堂的精华贴。

**【空空 mooc369】**：这门课是学软件的，为什么第 1 章很多内容是关于硬件的呀？

**【教师回复】**：初学者学习程序设计，首先应树立以下两个观念。

（1） 程序员向计算机硬件下达指令，然后由硬件执行指令。程序员应首先了解硬件的基本结构和工作原理，这样才能知道如何向硬件下达指令。

（2） 程序是一组下达给计算机硬件的指令序列（或称为语句序列）。仅从语法角度去理解，语句是抽象的。初学者要学会从有形的硬件去理解抽象的语句和语法。

程序设计课程只在一开始简单介绍一下硬件，后续章节在讲解 C++语法时会提及内存或 CPU 等硬件。

**【luckymooc360】**：C++语言为什么提出引用和指针的概念？

**【教师回复】**：程序员通过定义变量来申请内存，再用变量名访问所分配的内存单元。大型程序需要多个程序员协作开发，共同完成。如果其他程序员想访问上述变量的内存单元，例如读取其中的数据，可以吗？答案是肯定的，可以，但只能通过引用或指针来访问（即间接访问）。

**【小二上盘 ID】**：老师，我们学校先学 C 语言，然后学的是 C#和 Java。怎样才能学好程序设计？以后从事工作用 Java 好，还是 C/C++好？

**【教师回复】**：如何学习程序设计，这是很多初学者经常提出的问题。经过一段时间的学习，很多同学会产生新的困惑。例如，理论知识看了好几遍了，但是怎样提高编程能力？自己对 MOOC 课程的学习效果很满意，下一步该做什么？程序设计的学习过程是什么样的，最终能学到什么程度？针对这些问题我谈一下个人体会，供同学们参考。

学习程序设计大致可以分为三个阶段：初级、中级和高级。

（1）初级。初级阶段的目标是学习程序设计原理，其中包括计算机硬件基本结构及其工作原理，程序如何管理内存来存储数据（例如变量的定义与访问，数据类型，引用与指针等）、程序如何控制 CPU 来处理数据（例如各种不同的运算符，或通过控制语句来控制指令的执行顺序）等。程序设计原理还要学习如何设计大型、复杂的程序，这就需要学习程序设计方法。程序设计方法有两种，分别是结构化程序设计和面向对象程序设计。初级阶段学习结束后，同学们可以参加计算机等级考试（二级）或各种程序设计大赛等活动，并在应试过程中进一步提高自己的水平。

（2）中级。中级阶段的目标是学习程序应用开发（学会就能有好工作了哦）。程序应用开发需要基于别人的程序来开发，从零开始是不可能的。结构化程序设计方法规定：其他人给你函数库，你要会调用别人的函数。面向对象程序设计方法规定：其他人给你类库，你要知道如何使用别人的类库。因此我们在学习应用开发之前必须要掌握结构化程序设计方法或面向对象程序设计方法。目前面向对象程序设计是主流，已经很少有人继续给程序员提供函数库了，所提供的都是类库。掌握了面向对象程序设计方法，你只要拿到微软公司提供的 MFC 类库（随 Visual C 6.0 或 Visual Studio 提供），就可以用 C++语言开发 Windows 图形界面的程序（或者你拿到苹果公司的类库、谷歌公司的类库，就可以使用这些类库开发 iPhone 或 Android 系统的 App 了）。不同的操作系统是由不同厂家开发的，它们对计算机语言的支持程度有所不同。例如，Windows 操作系统是由微软开发的，开发 Windows 软件主要使用 C++和 C#语言；MacOS/iOS 操作系统是由苹果公司开发的，开发 MacOS/iOS 软件主要使用 Objective-C（C++的变种）；Android 操作系统（从 Linux 演变而来）由谷歌公司主导，开发 Android 软件主要使用 Java 语言。可以看出，程序应用开发可能会用到不同的计算机语言，但程序设计原理是共同的。

（3）高级。高级阶段的目标是提高自己的理论水平，不光要知其然，还要知其所以然。计算机专业的同学需要先学习程序设计原理，然后再学习计算机组成原理、数据结构、操作系统、编译原理、数据库原理、计算机网络、计算机图形学、数字图像处理、算法设计、离散数学和人工智能等课程。在学习完这些专业课程之后，你的理论水平会得到很大提高，将会成为一名真正的高手。

【LingDash】：老师，总地来说讲得挺好的，但是我有些想法。比如在讲类的继承与派生时，应先让学生体会到，每个类都从底层开发十分麻烦，那么怎么解决呢？这时引出“类的继承与派生”，将用与不用“类的继承与派生”进行比较，这样的学习会更加符合人的认知规律。

【教师回复】：有道理，接受你的建议，谢谢！

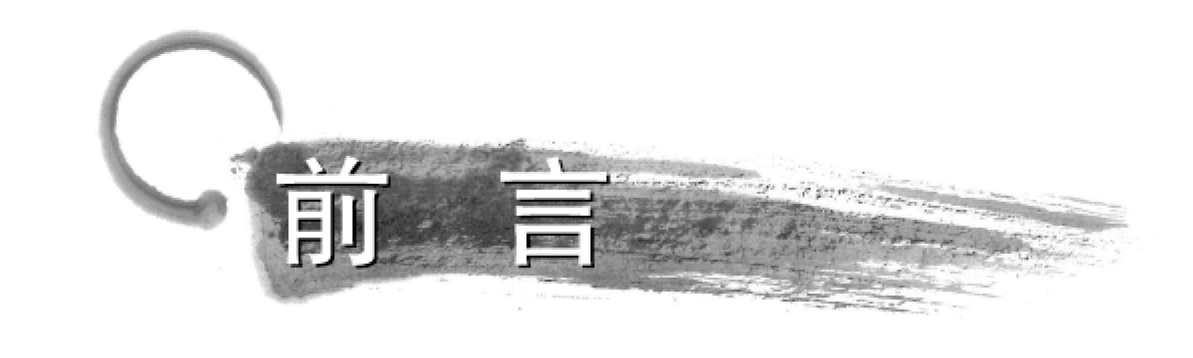
【cc76965】很高兴能够遇到这门公开课。这门公开课的条理十分清晰，让我对 C++的语法有了一个清晰的脉络。我是一名计算机专业的学生，已经学了数据结构，但是缺少开发项目的经验。不知道如何去寻找开发项目，也不知道如何下手，希望老师能够给我一些建议。

【网友 Crazy 峰少回复】：下面这个网站里有大量的编程题目可以做：https://www.patest.cn/。

这一版就是在总结广大 MOOC 学员所提出的难点、问题和建议的基础上对上一版所做的修订。重点完善了结构化程序设计和面向对象程序设计这两种程序设计方法的内容，使之更加系统化。

在此谨向中国大学 MOOC 和所有提出宝贵建议的 MOOC 学员们表示感谢！

作　者
2017 年 3 月于北京




### 1. 关于 MOOC

MOOC（Massive Open Online Course），即大规模开放在线课程，中文译为“慕课”，是近几年兴起的一种基于互联网的新型教学模式。2012 年被称为“MOOC 元年”。MOOC 教学模式实现了两个转变，即由以教师为中心向以学习者为中心转变，学习者则由被动学习向主动学习转变。与普通网络教学视频所不同的是，MOOC 实现了从授课到习题、讨论、答疑、测验，直到最终学习评价等环节的完整教学过程。与传统课堂授课不同的是，开设 MOOC 课程需重新梳理和组织知识点，并分别提供适合线上使用的练习题以及线下使用的讨论或实验题。

可以将线上 MOOC 与线下课堂这两种教学模式结合起来。线上 MOOC 就是先由学生自主完成知识学习，例如观看视频、做线上习题等。线下课堂则是由教师组织课堂讨论、实验、测验，或讲解重点疑难问题。“线上 MOOC，线下课堂”是对现有“课上听课，课下作业”教学模式的翻转。虽然 MOOC 教学模式尚处于起步试验阶段，但大多数网络学习者十分喜欢 MOOC。目前已涌现出很多知名的 MOOC 网站，例如国外的 Coursera、Udacity 和 Edx，国内的中国大学 MOOC、学堂在线和雨虹学网等。中国农业大学也正在基于雨虹学网积极开展校内课堂教学改革方面的尝试与探索。

学习 C++语言程序设计需要边阅读、边思考、边消化吸收。虽然有了大量的网上资源，但纸质教材仍是初学者线下学习的首选方式，这也是作者编写出版本书的目的。

### 2. 本书特色

1）适用于 MOOC 在线教育课程

本书按应用需求来梳理和组织 C++语言的知识点，其中包括结构化程序设计方法和面向对象程序设计方法。内容编排由易到难，循序渐进。每个小节都设计了适合在线评判的单选练习题，每章则设计了适合课堂讨论的程序阅读题、改错题和编程题。

2）采用案例教学

每个知识点都从精心设计的任务需求开始导入，然后提出对应的实现方法，最后系统地阐述其语法细则，既保证了知识体系的完整性，又能让读者直观理解抽象的概念和原理。

3）创新教学方法

本书从三个方面对 C++语言教学进行了探索。一是强化初学者对“程序由计算机硬件执行”这一基本概念的认知，从有形的硬件来理解相对抽象的程序，这样各种语法概念就不再那么抽象了；二是明确提出程序设计过程中存在不同的程序员角色，并充分利用角色

来引导读者理解语法的应用语境；三是明确提出必须从代码分类管理、数据类型、归纳抽象和代码重用等多个维度才能准确理解面向对象程序设计方法。教学实践表明，上述教学方法可降低学习难度。

3. 内容摘要

本书内容按章节顺序可分为三部分，分别是程序设计基础（第 1~4 章）、结构化程序设计方法（第 5~6 章）和面向对象程序设计方法（第 7~10 章）。

第 1 章　程序设计导论。从初学者对计算机已有的认知开始，将初学者逐步引导到计算机程序的世界。本章首先介绍计算机硬件、指令及机器语言、程序等基本概念，然后描述程序与计算机硬件、程序员、用户之间的关系，让读者在一开始就能明确程序员的职责，实现从用户到程序员的角色转换。本章要点：一是让读者从有形的硬件来理解相对抽象的软件；二是让读者认识到计算机中的数据是有类型的，类型决定了数据在计算机中的存储位数和存储格式；三是让读者知道，学习程序设计和学习编程语言不是一回事。和 C 语言、Java 语言相比，C++语言的知识体系更加系统、全面。本书选用 C++语言作为程序设计初学者的入门语言。

第 2 章　数值计算。本章从最简单的数值计算问题开始，以案例教学的方式让读者领会程序设计中一些最基础的概念，其中包括程序中的变量和常量、表达式与运算符、数据的输入和输出等。最后介绍了 C++程序访问内存的三种方式，它们分别是变量名、引用和指针。本章要点：一是让读者将程序中的数据与内存联系起来，这样就很容易理解数据类型、引用和指针等初学者难以掌握的概念；二是让读者重点关注运算符的运算规则、优先级和结合性等语法细节；三是让读者初步体会到计算机语言与人类语言的不同之处，即计算机语言的语法规则非常严格，甚至到了机械的程度，稍有不慎就会出现语法错误。

第 3 章　算法与控制结构。本章讲解程序中的算法及三种算法基本结构，并通过选择结构和循环结构中的条件引出布尔类型。C++语言通过选择语句来描述选择结构算法，通过循环语句来描述循环结构算法。最后通过案例简单讲解算法的设计与评价方法。本章要点：一是让读者了解绝大部分复杂算法都可以由三种基本的算法结构来完成；二是让读者掌握布尔类型的作用及其相关的运算符；三是让读者了解编程能力实际上是一个人计算思维能力的反映，阅读程序和模仿编程是初学者培养计算思维能力的两个重要途径；四是让读者根据案例认真体会如何根据算法合理选用不同的控制语句。

第 4 章　数组与文字处理。本章学习如何在程序中存储和处理大量数据。数组可以存储大量具有相同类型的数据集合。计算机只能存储和处理数值数据，而文字处理程序所处理的对象是字符数据，为此，C++语言引入了字符类型。读者需深入了解字符编码和字符类型。文字处理必须使用数组，即字符型数组。本章最后用一节的篇幅简单介绍了中文处理及 Unicode 编码。本章要点：一是让读者重点掌握数组定义及访问的语法规则；二是让读者认识到计算机内部对数组的管理和访问是通过指针（即内存地址）来实现的；三是让读者通过具体案例初步了解数组的常用处理算法。

经过前 4 章的学习，读者已掌握了程序设计原理基础部分的内容。那么该如何编写更大型的计算机程序呢？这就需要进一步学习程序设计原理的高级部分，即程序设计方法。

程序设计方法的基本思想是：将大型程序中的数据和算法分解成程序零件，将不同零件的设计任务交由不同的程序员完成，这样就能以团队的形式来共同开发。更进一步，如果所分解出的程序零件在以前项目中曾经开发过，或者可以从市场上购买到，那么就可以直接使用这些零件来组装软件，实现快速开发。使用已有的程序零件，实际上是重用其程序代码，这就是程序设计中的代码重用。从第 5 章开始，本书对程序设计方法进行系统介绍。

第 5 章　结构化程序设计之一。本章学习如何将一个复杂的数据处理算法分解成多个简单模块，分而治之，这被称为是结构化程序设计方法。C++语言支持结构化程序设计方法，以函数的语法形式来描述和组装模块，即函数的定义和调用。函数是结构化程序设计方法的基础，它为代码重用提供了有效的手段。函数之间需要共享数据才能完成规定的数据处理任务。C++语言提供了集中管理和分散管理两种不同的数据管理策略。本章要点：一是读者要准确领会结构化程序设计的思想内涵，并熟练掌握 C++语言中函数相关的语法知识；二是让读者深入计算机内部，了解程序执行时其代码和变量在内存中的存储原理，这样可以更容易理解变量作用域和生存期等抽象的概念；三是读者要准确把握函数间传递数据的三种方式；四是读者要分别站在两种不同的程序员角度，即定义函数的程序员和调用函数的程序员，才能更容易地理解函数相关的各种语法知识。

第 6 章　结构化程序设计之二。本章学习如何以多文件结构的形式来组织和管理源代码，并介绍几种常用的编译预处理指令；然后再介绍几种特殊形式的函数，其中包括带默认形参值的函数、重载函数、内联函数、带形参和返回值的主函数以及递归函数等。本章还会介绍与 C 语言相关的系统函数和自定义数据类型。本章最后以微软公司开发的 Win32 API 函数库为例介绍如何开发一个 Windows 图形用户界面程序，并对结构化程序设计方法进行简单的回顾和总结。本章要点：一是学习掌握与多文件结构相关的语法知识，其中包括外部函数和全局变量的声明、头文件等；二是重点掌握带默认形参值的函数、重载函数和内联函数这三种常用的特殊函数形式；三是牢固树立重用代码的思想，学会通过调用他人编写的函数来提高开发效率。

第 7 章　面向对象程序设计之一。面向对象程序设计方法将程序中的数据元素和算法元素根据其内在关联关系进行分类管理，这就形成了“类”的概念。分类可以更好地管理。类相当于是一种自定义的数据类型，用类所定义的变量称为“对象”。本章通过具体案例演示了结构化程序设计是如何演变到面向对象程序设计的，然后再系统地介绍面向对象程序设计方法。本章内容包括类的定义、对象的定义与访问、对象的构造与析构、类中的常成员与静态成员以及类的友元等。本章要点：一是读者必须从代码分类管理、数据类型、归纳抽象和代码重用等多个维度才能准确理解类与对象的概念；二是读者需认真学习类与对象编程的具体语法规则；三是深入领会面向对象程序设计通过设置访问权限来实现类封装的基本原理；四是深入了解对象的构造与析构过程，程序员通过编写构造与析构函数来参与对象的构造与析构过程；五是读者要懂得从两个不同的角度，即定义类的程序员和使用类定义对象的程序员，才能更容易地理解类与对象相关的各种语法知识。

第 8 章　面向对象程序设计之二。重用类代码有三种方式，分别是用类定义对象、类的组合和类的继承。本章讲解类的组合与继承。程序员可以基于已有的零件类来定义新的整体类，这就是类的组合。程序员可以继承已有的基类来定义新的派生类，这就是类的继

承与派生。利用派生类和基类之间的特殊关系可以进一步提高程序代码的可重用性，这就是面向对象程序设计中的对象替换与多态技术。本章将具体讲解与多态相关的运算符重载、虚函数和抽象类等概念。最后本章将简单讨论一下类的多继承。本章要点：一是让读者学会使用组合和继承的方法来定义新类，这样可以提高类代码的开发效率；二是读者应理解，类在组合或继承时可以进行二次封装；三是从提高程序代码重用性的角度可以更容易地理解对象多态性；四是多继承会导致语法陷阱，新的面向对象程序设计语言（例如 Java 和 C#）已不再支持类的多继承，而只支持接口的多继承，读者只需要了解多继承的基本原理即可。

第 9 章　流类库与文件 I/O。C 语言通过输入/输出函数（例如 scanf 和 printf）实现了数据的输入和输出。C++语言则是通过输入/输出流类为程序员提供了输入/输出的功能。这些输入/输出流类都是从类 ios 派生出来的，组成了一个以 ios 为基类的类族，这个类族被称为 C++语言的流类库。本章将介绍流类库中三组不同功能的输入/输出流类，它们分别是通用输入/输出流类、文件输入/输出流类和字符串输入/输出流类。本章要点：一是读者应理解之前所用的 cin、cout 指令实际上分别是通用输入/输出流类的对象；二是通过本章学习，读者可以从侧面了解全球顶尖的 C++程序员是如何来设计和编写类的，这样可以帮助读者进一步深入体会前面所学习的各种面向对象程序设计知识；三是重点学习如何进行文件读写操作，大部分程序都需要使用文件来保存数据。

第 10 章　C++标准库。C++语言全盘继承了 C 语言的标准 C 库，另外又增加了一些新的库。新库中包含一些新增的系统函数，但更多的是为面向对象程序设计方法所提供的系统类库，这些新库被统称为 C++标准库。为了更好地凝练源代码，C++语言引入了模板技术，其中包括函数模板和类模板。模板技术是一种代码重用技术，C++标准库在编写时就采用了模板技术，因此标准库能以较少的代码量来提供很强大的功能。本章内容重点介绍模板技术、C++语言的异常处理机制以及 C++标准库所提供的数据集合存储及处理功能。本章最后以微软公司开发的 MFC 类库为例介绍如何开发一个 Windows 图形用户界面程序。本章要点：一是让读者了解如何使用模板技术来提高函数和类代码的可重用性；二是重点学习 C++语言的异常处理机制；三是初步掌握如何使用 C++标准库中的向量类、列表类、集合类和映射类来存储和处理数据集合。

学习完 C++面向对象程序设计之后，程序员在拿到一个具体的程序设计任务时，首先应当考虑有哪些现成的类库可以使用。使用现成的类库开发程序，开发周期将大大缩短。基于已有的类库开发程序，相当于是用别人已经做好的零件来组装产品。程序的应用开发，通常就是用已有的程序零件来组装自己的软件产品。只要掌握了面向对象程序设计方法和 C++语言，相信每位读者都能够借助各种第三方类库，发挥出无限的开发潜能。

#### 4. 使用建议

凡希望开设 C++语言程序设计在线教育课程的教师，可将本书作为授课教材。联系作者可免费获得配套教学课件。参加在线课程学习的学生可将本书作为线下阅读教材。

如将本书作为课堂教学用书，则建议讲课学时和实验学时各为 32 学时，合计 64 学时。每学时 50 分钟。作者本人按如下方式安排讲课学时：第 1、3、4、9、10 章各 2 学时，第

2、5、6、7 章各 4 学时，第 8 章 6 学时。

联系作者：kandaohong@cau.edu.cn

5. 致谢

作者编写本书的想法源于中国农业大学“雨虹学网：面向主动学习的教学云平台建设”项目。在参与相关系统开发和教学实践的过程中，作者积累了一些 MOOC 在线课程教学的经验。

本书编写过程中，得到了中国农业大学信电学院高万林院长的热情鼓励和大力支持。本书部分素材来自于雨虹学网的教学实践活动，这得益于张晓东教授、孙瑞志教授等领导的关心和指导。另外，本书在编写过程中还得到了郑立华、吕春利、冀荣华、刘云玲、陈瑛、周绪宏、胡慧、段晶洁、李鑫、李静等同事和同学的热心帮助。在此一并致以衷心的感谢！

最后，感谢家人对我的理解和支持。

作　者

2015 年 9 月于北京

# 目 录

# 第 1 章 程序设计导论

**计算机**是一种能够按照指令完成数值计算的机器。**指令**由人（称为程序员）下达，由计算机中的电子电路（称为硬件）识别和执行。计算机硬件能够识别和执行的指令集合称为**机器语言**。机器语言应尽量简单，以提高执行速度，同时降低硬件生产成本。

**程序员**可以将多条指令编排成一个指令序列（称为**程序**），一次性提交给计算机，由计算机自动按顺序连续执行（如图 1-1 所示）。程序描述了某种数据处理的过程和步骤。程序可以重复执行，每执行一次程序，计算机就完成一次程序所规定的数据处理过程。

一台可以工作的计算机由硬件和软件两部分组成，因此被称为一个**计算机系统**。**硬件**包括中央处理器、存储器和输入/输出设备等。**软件**包括操作系统、数据库管理系统、软件开发工具以及各种不同功能的应用程序。**用户**可以使用计算机系统协助自己完成某种特定工作，其操作过程通常是：首先启动计算机，执行某个程序，然后按照程序提示选择功能或输入原始数据，最后查看程序的处理结果（如图 1-2 所示）。程序由程序员编写。用户应购买或通过合法渠道获得程序，并预先安装到自己的计算机系统中。

学习程序设计，就是让自己从使用程序的用户角色提升到编写程序的程序员角色。

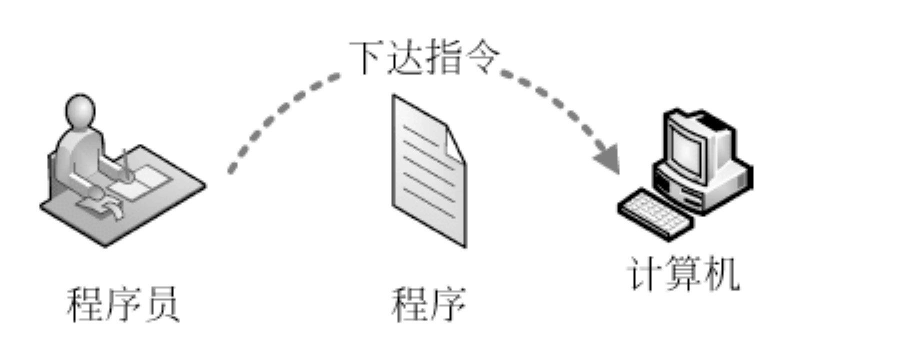



图 1-1　程序员、计算机与程序

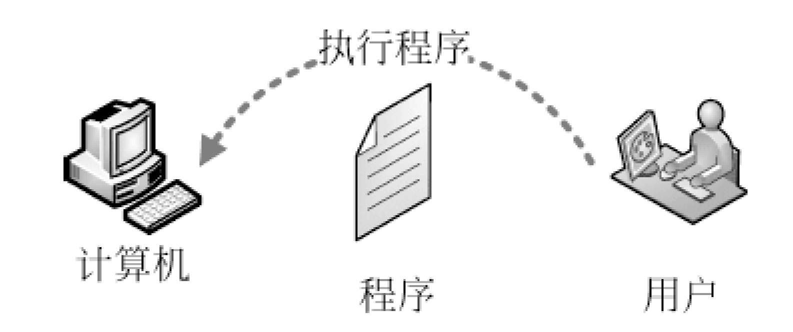



图 1-2　用户、计算机与程序

## 1.1　计算机硬件结构

冯·诺依曼体系结构（图 1-3）是所有现代计算机（例如台式机、笔记本计算机、智能手机等）硬件结构的基础。其核心思想是**存储程序计算机**，即包含一组指令序列的程序预先存储在存储器中，执行时由中央处理器从中读取指令，并按顺序自动连续执行。存储程序计算机思想的最大贡献是将计算机变成了一种在程序控制下自动工作的机器，是从手动到自动的跨越。为了便于硬件实现，冯·诺依曼体系结构明确提出采用**二进制**数制对数据进行存储和运算，同时，用二进制数值编码来表示不同的机器语言指令。编码后，一个机器语言的程序实际上就是一个二进制数值编码序列，可以直接被计算机硬件识别、执行。

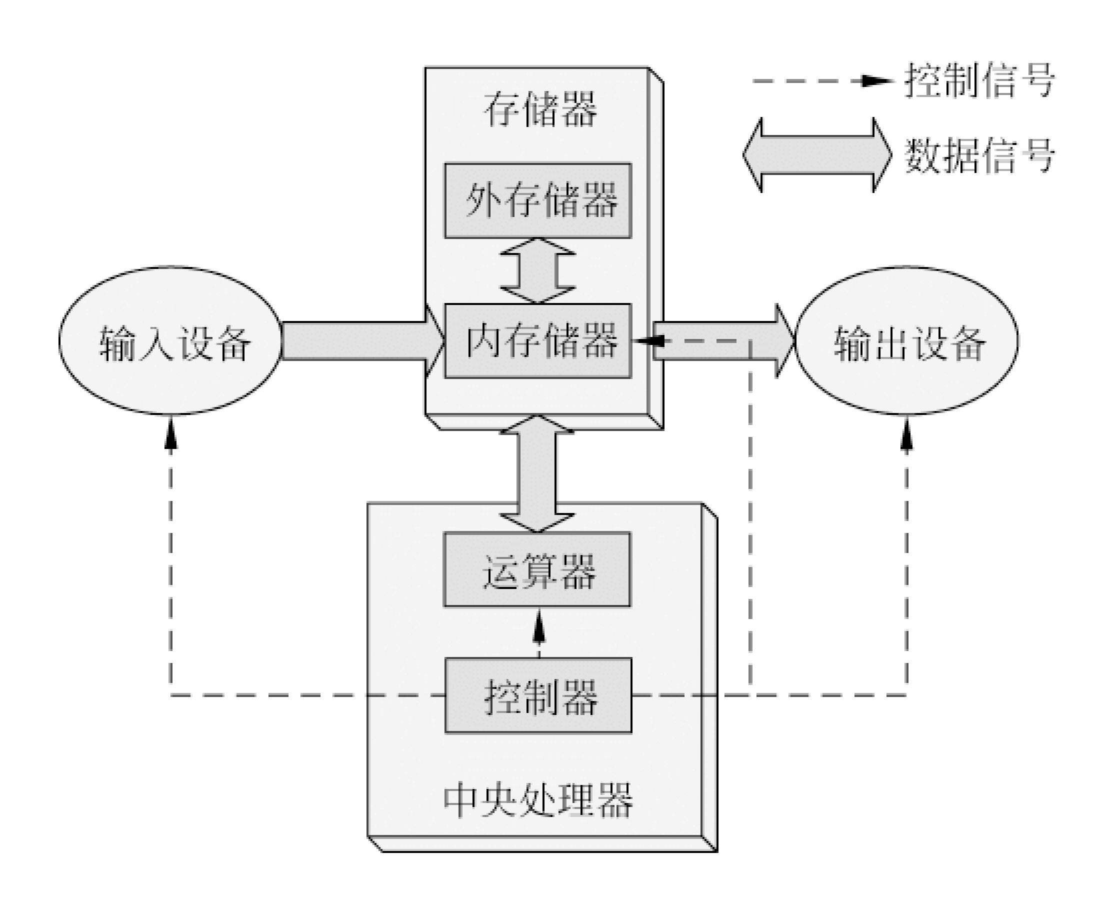



图 1-3 冯·诺依曼体系结构

冯·诺依曼体系结构包含 5 个基本组成部分：**运算器**、**控制器**、**存储器**、**输入设备**和**输出设备**。运算器和控制器合起来被称为**中央处理器** CPU（Central Processing Unit）。存储器可分为内存储器（简称内存）和外存储器（简称外存，例如硬盘）两种。输入设备包括键盘、鼠标、扫描仪、麦克风等，而输出设备则包括显示器、打印机、音箱等。输入/输出设备统称为计算机的外围设备，是人与计算机进行交互的接口。

### 1．存储器

程序预先存储在存储器中。执行程序时，CPU 从存储器中逐条读出指令，识别并执行该指令，然后再到存储器读取下一条指令。存储器为 CPU 连续执行指令，实现自动工作奠定了基础。

程序是用来处理数据的。程序中的数据包括原始数据、中间结果、最终结果等。数据要存放在存储器中才能被 CPU 读取和处理，CPU 处理后的结果也只能保存回存储器中。

存储器可分为**内存**和**外存**两种。内存的读写速度快，但造价高，而且内存中的数据在断电时（例如关机）会丢失。外存（例如硬盘）读写速度慢，但造价低，而且断电时数据不丢失，可以长期保存。程序平时以文件形式存放在外存上，执行时才被读入内存，并在内存中执行。程序所处理的数据也需要放在内存才能被处理，例如硬盘上的一份 Word 文档，打开文档时被读入内存处理，处理完后保存文档，重新将文档写回硬盘以便长期保存。

对存储器的操作有**写入**（write）和**读出**（read）两种，即将数据写入存储器，或从存储器读出数据。对存储器的读写操作被统称为对存储器的**访问**（access）。

1 个二进制**位**（bit）可以存储 1 个二进制数，即 0 或 1。8 个二进制位组成 1 个**字节**（byte）。其他常用的存储单位还有 KB（$2^{10}$=1024 字节）、MB（$2^{20}$=1024KB）、GB（$2^{30}$=1024MB）等。

对存储器进行读写操作的最小单位是字节，一个字节被称为是一个存储单元。计算机内存包含有很多个存储单元。为每个存储单元指定一个整数编号（通常从 0 开始，连续编号），称为存储单元的**内存地址**。程序通过内存地址将数据写入某个存储单元，或将某个存储单元中的数据读出来。

### 2．中央处理器

中央处理器 CPU 是计算机的“大脑”，包含运算器和控制器。CPU 执行一条指令的过程分为读取指令、识别指令和执行指令 3 步，这个过程被称为一个“取指-译码-执行”周期。**控制器**负责从内存中读取指令，并识别出指令的类型。**运算器**负责执行指令，实现算术运算（例如加、减、乘、除）、关系运算（例如比较两个数的大小）等运算。

CPU 内部还包含一个小容量、高速度的存储器。该存储器由多个被称作**寄存器**的存储单元组成。CPU 执行程序中的某条指令时，首先将存储在内存中的指令和数据读入寄存器，然后再识别、执行。执行结果也先暂存在寄存器，然后再写入内存。

### 3．输入设备

输入设备是人向计算机下达指令和输入信息的装置。输入设备主要包括键盘（字符输入设备）、鼠标（图形输入设备）、扫描仪（图像输入设备）、麦克风（声音输入设备）等。输入设备将所接收到的用户操作（例如用户按压键盘按键、单击鼠标左键等），或不同种类的输入信息（例如扫描仪中的纸质文件或照片、麦克风接收到的声音等）统统转换成数值形式的数据，然后再存放到内存中，交由程序进行处理。

计算机只能存储、处理数值形式的数据，即只能进行数值计算。任何信息，只有在转换成数值形式的数据（称为**数字化**）之后才能交由计算机进行处理。输入设备就是计算机对信息进行数字化的设备。只要有相应的数字化设备，任何信息都可以使用计算机来进行处理。例如，目前正在研究一种称作“电子鼻”的输入设备，它能够对气味进行数字化，然后识别出气味种类。

### 4．输出设备

输出设备是将计算机的信息处理结果反馈给人的装置。输出设备主要包括显示器、打印机、音箱等。不同种类信息在计算机中都是以数值形式进行存储和处理的。输出时，需要将信息还原成原来的形式，即人可以接收的形式。例如，输出字符、图形或图像时需通过显示器或打印机还原成视觉信息，输出声音时需通过音箱或耳机还原成听觉信息。

存储器、中央处理器和输入/输出设备等部件通过一组平行导线（称为**总线**）连接在一起（图 1-4）。各个部件之间通过总线进行通信，传递地址、数据和控制信号等信息。

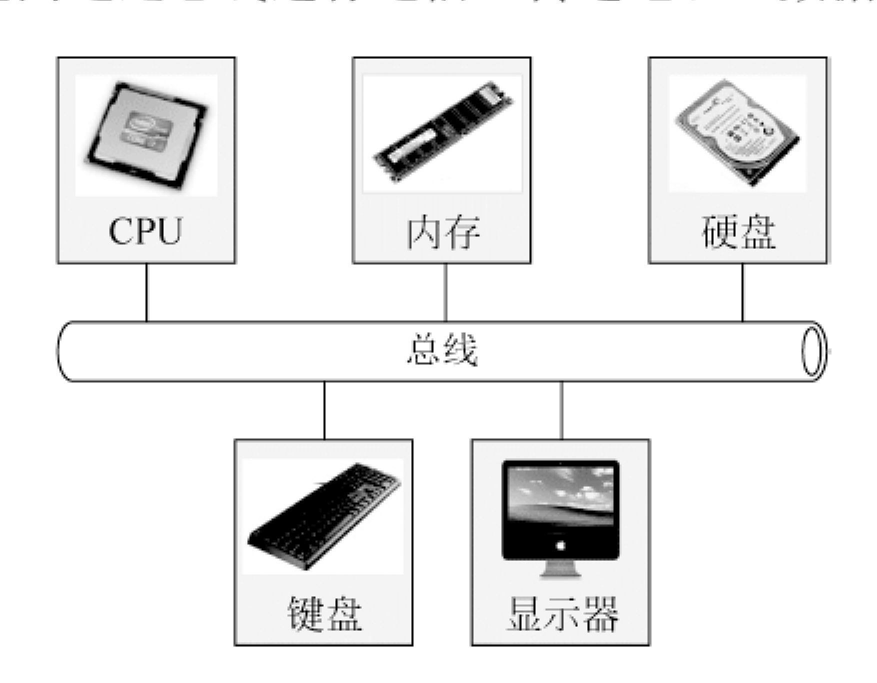



图 1-4　计算机的总线结构

## 本节习题

1．计算机硬件结构中，负责识别并执行指令的部件是（　　）。
A．鼠标　B．硬盘　C．主板　D．CPU
2．计算机硬件能识别并执行下列哪种语言所表达的指令？（　　）
A．英语　B．中文　C．机器语言　D．C++语言
3．CPU 只能处理存放在（　　）中的数据。
A．硬盘　B．内存　C．U 盘　D．光盘
4．触摸屏属于什么类型的设备？（　　）
A．输入设备　B．输出设备
C．输入＋输出设备　D．存储设备
5．下列哪种设备不具备数字化（即将信息转换成数值数据）的能力？（　　）
A．音箱　B．麦克风　C．扫描仪　D．键盘

# 1.2 计算机程序

设想一下，如何基于 1.1 节讲解的硬件结构让计算机帮助人类做一些事情，比如将摄氏温度转换成华氏温度。为了解决这个问题，首先应当由程序员编写一个程序，该程序用计算机语言来描述温度换算的步骤和方法，然后计算机按照程序执行相应的操作。执行这个温度换算程序，计算机将提示用户输入摄氏温度，然后按照程序所描述的算法进行温度换算，并将换算得到的华氏温度反馈给用户。

### 1．计算机程序

计算机**程序**（program）是使用某种**计算机语言**编写的一组指示计算机进行数据处理的指令序列。使用计算机处理数据一般可分为 4 个步骤。

**申请内存空间**。数据要存放在内存中才能被 CPU 读取和处理，处理后的结果也只能保存回内存中。程序需要通过申请内存空间指令，预先为数据分配好内存单元。数据包括原始数据、中间结果和最终结果等。

**输入原始数据**。计算机通过输入设备输入原始数据。程序通过输入指令将原始数据输入到预先分配好的内存单元，等待处理。键盘是最常用的输入设备，可以输入数值、文字等数据。

**数据处理**。CPU 负责数据处理。它从内存中读取原始数据，将处理结果再放回内存。程序通过由不同运算符构成的表达式来对数据进行处理。

**输出处理结果**。数据处理结束后，应当将处理结果通过输出设备反馈给用户。程序通过输出指令将存放在内存中的处理结果送往输出设备。显示器是最常用的输出设备，可以显示数值、文字、图形、图像等数据。

最开始，程序员是使用**机器语言**来编写程序的。程序员需要熟记机器语言中每条指令

的二进制编码，可以想象其编写程序的难度。后来，人们使用助记符来代替指令的二进制编码，例如用 ADD 表示加法、用 SUB 表示减法，这种以助记符来表示指令的语言被称为**汇编语言**。汇编语言提高了程序员编写程序的效率。再后来，人们又发明了更容易学习和使用的计算机语言，这就是今天我们常用的**高级语言**。机器语言、汇编语言、高级语言都属于计算机语言。C++语言是一种高级语言，例 1-1 给出一个用 C++语言编写的温度换算程序，其中包含了三部分内容，分别是一些注释、一组指令序列以及一些 C++语言规定的程序格式。

**例 1-1　一个用 C++语言编写的温度换算程序**

```
1   /*
2      一个 C++程序实例：
3      将摄氏温度换算成华氏温度
4   */
5   #include <iostream>
6   using namespace std;
7
8   int main( )
9   {
10      double ctemp, ftemp;            //申请内存空间
11      cin >> ctemp;                   //从键盘输入摄氏温度
12      ftemp = ctemp * 1.8 + 32;       //温度换算
13      cout << ftemp;                  //在显示器上输出华氏温度
14      return 0;                       //程序结束，返回操作系统
15  }
```

（1）**注释**：使用自然语言（例如中文或英文）对程序代码进行说明，以便程序员今后阅读或修改程序。

第 1~4 行：多行注释形式，以“/*”开头、“*/”结束。该段文字解释程序的功能，即“将摄氏温度换算成华氏温度”。

第 10~14 行的后半部分：单行注释形式，以“//”开头的文字说明，到行尾结束。

（2）**指令序列**：第 10~14 行，程序的正文部分，即一组指令。

第 10 行：申请内存空间。定义 2 个保存温度数据的变量 ctemp 和 ftemp，分别用于保存摄氏温度和华氏温度。double 说明温度数据为实数类型。执行该指令，计算机为所定义的变量分配内存单元，用于保存数据。

第 11 行：输入原始数据。cin 指令从键盘输入数据并保存到变量 ctemp 中。执行该指令，计算机等待用户在键盘上输入摄氏温度，所输入的数据将被存放到变量 ctemp 所分配的内存单元中。

第 12 行：数据处理。使用换算公式“华氏温度=摄氏温度×1.8 + 32”将变量 ctemp 中存放的摄氏温度换算成华氏温度。通过“=”将换算结果保存到变量 ftemp 中。

第 13 行：输出处理结果。cout 指令将变量 ftemp 中所保存的华氏温度在显示器上显示出来。

第 14 行：结束程序。return 被称为返回指令。执行该指令，计算机将结束当前程序的运行，返回操作系统。

（3）**C++语言规定的程序格式**：用 C++语言编写程序有一些规定的程序格式，每个C++程序都应该遵守这些规定。例如：

第 5~6 行：声明导入某些外部程序。

第 8 行：定义主函数“int main( ) { 指令序列 }”。每个 C++程序都有一个包含指令序列的主函数 main。主函数用一对大括号“{ }”将指令序列括起来。

第 10~14 行：指令序列中的每条指令称为C++程序的一条语句。语句以分号“;”结尾。

## 2．程序的执行

程序由**程序员**编写，由**计算机**执行。程序平时是以文件形式保存在硬盘上的，执行时被读入内存，在内存中建立一个副本（图 1-5）。CPU 从程序主函数中的第一条指令开始，依次执行指令序列，直到最后一条指令，或遇到返回指令 return 时结束执行。程序执行时，计算机会为程序中所定义的变量分配内存空间，例如图 1-5 中的 ctemp 和 ftemp。程序执行时，CPU 会按照程序中的指令进行运算，例如执行例 1-1 第 12 行的“ftemp = ctemp * 1.8 + 32;”指令时，CPU 会从内存中读出 ctemp 内存单元的数值进行运算，再将运算结果写入 ftemp 的内存单元。程序执行结束后，计算机将收回程序副本及其变量所占用的内存空间，以便于执行其他程序时使用。计算机每执行一次例 1-1 的程序就完成一次将摄氏温度换算成华氏温度的过程。

程序可以**复制**给多个用户。用户在自己的计算机上安装程序，然后执行（或称为启动）该程序。用户每执行一次程序就是使用了一次该程序的功能。用户只要安装了该程序，今后就可以随时执行，重复使用该程序的功能。

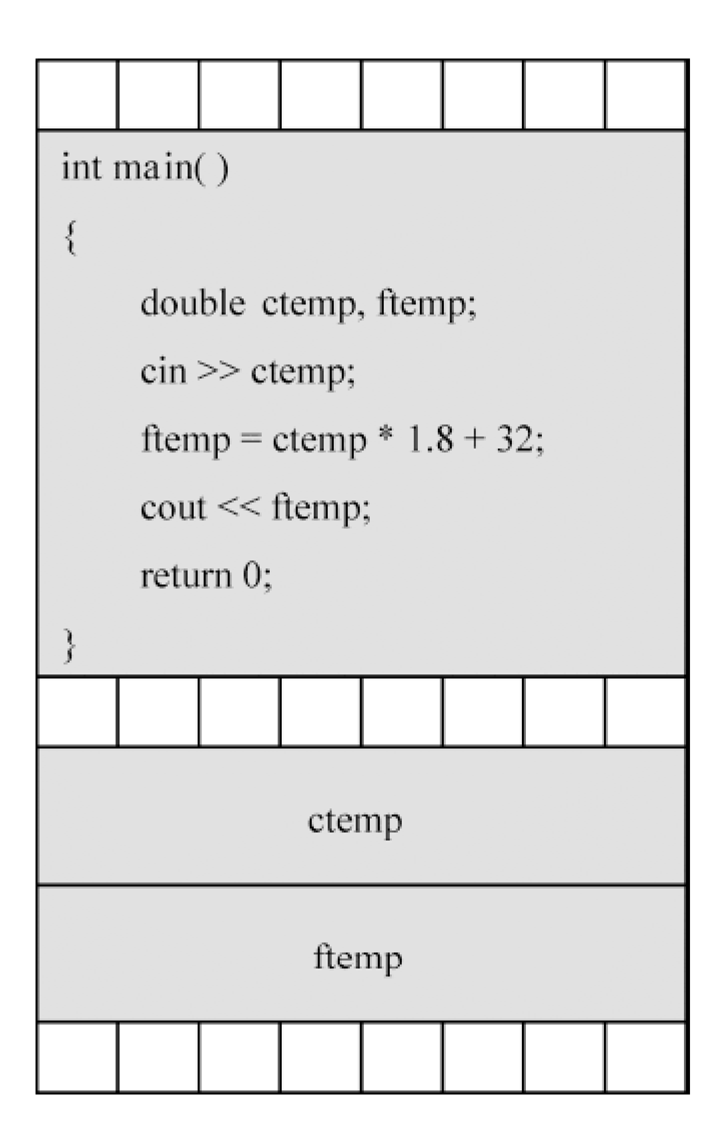



图 1-5 程序执行时的内存示意图

## 3．程序=数据+算法

程序是一组指令序列，描述了某种数据处理的过程和步骤。

**数据**是程序处理的对象。程序中的数据包括原始数据、中间结果、最终结果等。数据要存放在内存中才能被 CPU 读取和处理，处理后的结果也只能保存回内存中。如何根据所处理的数据来合理地使用和管理内存是编写程序的**第一项**工作内容。计算机程序通过定义和删除变量来申请、释放内存空间。为了进行温度换算，例 1-1 定义了 2 个变量 ctemp 和 ftemp 来分别存放摄氏温度（原始数据）和华氏温度（最终结果）。程序执行结束后，上述 2 个变量被自动删除，其所占用的内存空间被计算机收回。

将数据处理的过程细分成一组严格的操作步骤，这组操作步骤被称为**算法**。如何设计数据处理算法是编写程序的**第二项**工作内容。计算机程序通过选择不同功能的指令，并合理编排这些指令的顺序来实现算法。例 1-1 将温度换算的算法分解成 5 步完成。

（1）定义变量（申请内存空间）；

（2）输入需换算的摄氏温度；

（3）根据换算公式将其换算成华氏温度；

（4）显示换算结果；

（5）算法结束，返回。

数据和算法是一个程序应当包含的两项主要内容。可以简单地说，程序=数据+算法。

#### 4．程序的用户界面

程序执行过程中，通常需要用户输入原始数据或选择功能（称为输入），程序将计算得到的中间结果和最终结果反馈给用户（称为输出）。用户与程序之间的输入和输出操作统称为人机交互。目前，人机交互的形式主要有两种：命令行界面（Command Line Interface，简称 CLI）和图形用户界面（Graphical User Interface，简称 GUI）。

**1）命令行界面 CLI**

早期，用户操作计算机是通过键盘输入指令（或称为命令），计算机接收指令并执行对应的程序。这种操作程序的形式被称为命令行界面（图 1-6）。使用命令行界面的程序需要用户记忆相关的操作命令，这适用于承担系统维护工作的专业技术人员。

**2）图形用户界面 GUI**

图形用户界面的程序提供窗口、按钮、菜单等图形操作界面，用户通过指针设备（例如鼠标、触摸屏等）选择程序功能，操作程序。这种操作程序的形式被称为图形用户界面（图 1-7）。操作图形用户界面时，用户不需要记忆操作指令，简单易学，适用于广大的普通用户。

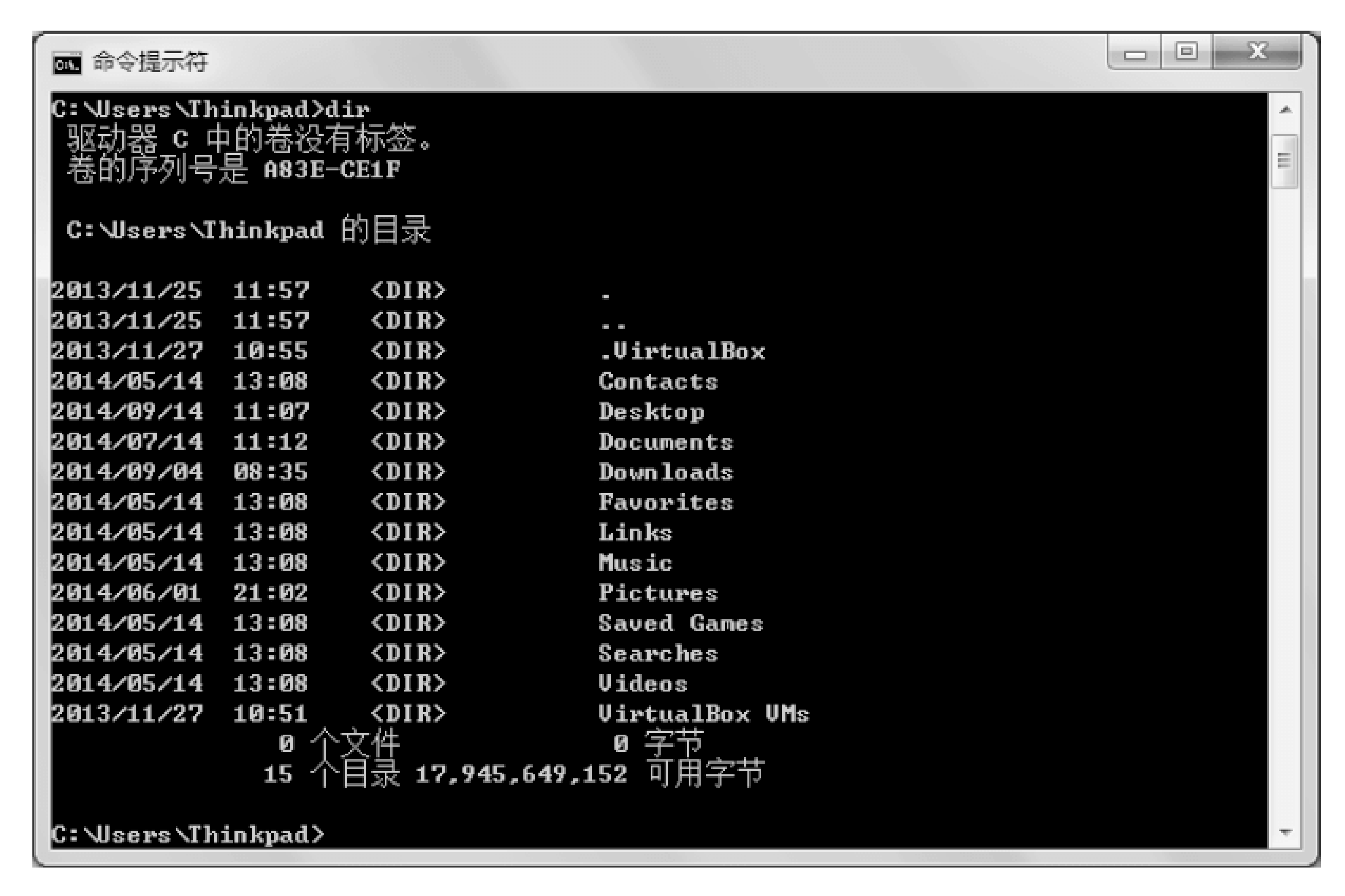



图 1-6　命令行界面 CLI




图 1-7　图形用户界面 GUI

## 本节习题

1．编写一个处理数据的计算机程序，通常第一步需要做什么？（　　）

A．申请内存空间　　B．输入原始数据
C．处理数据　　D．输出处理结果

2．下列哪种语言不属于计算机语言？（　　）

A．机器语言　　B．汇编语言　　C．高级语言　　D．英语

3．CPU 只能执行存放在什么地方的程序？（　　）

A．硬盘　　B．内存　　C．U 盘　　D．光盘

4．运行命令行界面程序的计算机必须配备下列哪种输入设备？（　　）

A．键盘　　B．鼠标　　C．触摸屏　　D．手写笔

5．在计算机内部，从键盘输入的数据首先被送往哪里？（　　）

A．运算器　　B．控制器　　C．内存　　D．外存

# 1.3　计算机程序开发

计算机程序的开发过程可粗略地分为设计、实现、测试和发布 4 个阶段。参与程序开发过程的人员统称为程序开发人员，或简称为**程序员**。

## 1.3.1　程序设计

程序设计按时间顺序还可细分为需求分析和程序设计两个阶段，即先进行需求分析，再根据需求进行程序设计。

在软件工程中，**需求分析**指的是：在建立一个新的或改变一个现存的计算机软件系统时，为描述新系统的目的、范围、定义和功能等所要做的全部工作。需求分析要对程序的用户单位进行调研和分析，弄清楚其功能和性能要求，包括需要输入什么数据，要得到什么结果以及最后应输出什么等。需求分析结束后，应该形成书面的需求分析报告。需求分析报告是下一阶段进行程序设计的基础和依据。

有了需求分析报告，程序开发人员即可进入**程序设计**阶段。程序设计可采用结构化程序设计方法，或面向对象程序设计方法。

**结构化程序设计**方法也称为面向过程的程序设计方法。结构化程序的设计方法就是：首先将一个求解复杂问题的过程划分为若干个子过程，每个子过程完成一个独立的、相对简单的功能；用算法来描述各个过程的操作步骤，每个算法称为一个**模块**。结构化程序设计采用“自顶向下，逐步细化”的方法来分解和设计算法模块，然后通过**调用**关系将各个模块组织起来，最终形成一个求解问题的完整流程。采用结构化程序设计方法，程序员重点考虑的是如何分解和设计算法。结构化程序设计方法一般使用流程图来记录算法设计结果。以特定的图形符号加上文字说明来表示算法中操作步骤及其顺序的图被称为**流程图**或**框图**。美国国家标准学会 ANSI（American National Standard Institute）规定了流程图的常用符号。我国的国家标准 GB 1526—89 与该标准基本相同。图 1-8 给出一个温度换算算法的流程图实例。

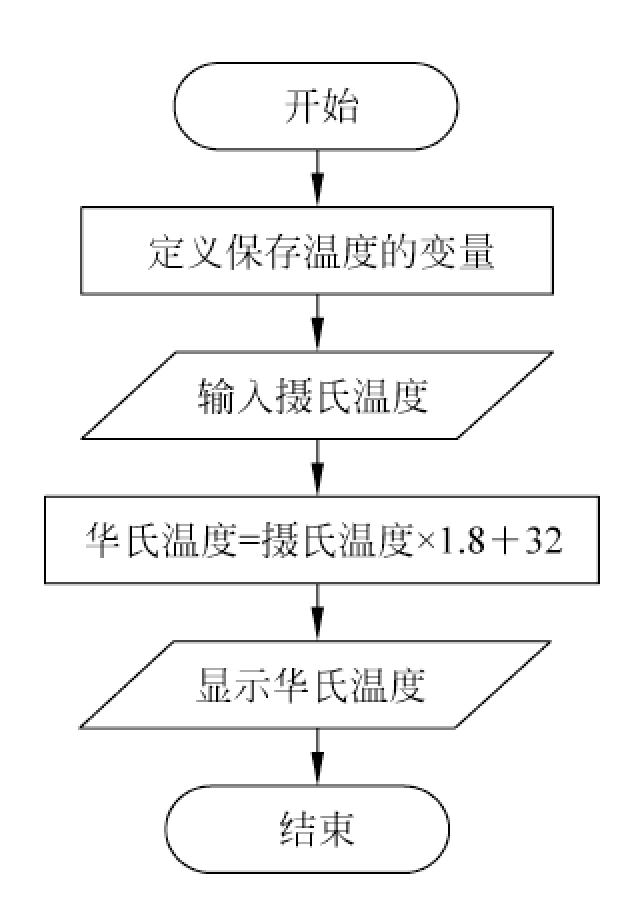



图 1-8　温度换算算法的流程图实例

**面向对象程序设计**方法是将待处理问题中的客观事物当做一个个独立的处理对象（称为**客观对象**），以归纳分类的思想把具有相似特性的对象抽象成**类**。类是程序设计中描述客观事物的数据模型，其中包括事物的**属性**（数据）和处理**方法**（算法）。按照类模型将客观对象定义成计算机中的对象（称为**内存对象**，或简称为**对象**），这样就可以交由计算机来处理了。通过“消息驱动”机制将各个对象组织起来，最终形成一个完整的计算机程序。面向对象程序设计的重点是类和对象的设计。面向对象程序设计方法一般使用**统一建模语言** UML（Unified Modeling Language）来记录类和对象等的设计结果。UML 使用 5 种不同的图来描述这些设计结果，它们分别是：

- 用例图(use case diagram)；
- 静态图(static diagram)，包括类图、对象图和包图；
- 行为图(behavior diagram)，包括活动图、状态图；
- 交互图(interactive diagram)，包括顺序图、协作图；
- 实现图(implementation diagram)，包括构件图、部件图。

这些图从不同的侧面对所设计的程序进行描述。图 1-9 给出温度换算问题中一个温度类 Temperature 的类图及其对象 t 的对象图例子。

| Temperature |
| --- |
| + ctemp : double |
| + SetCTemp(double t) : void<br>+ GetCTemp( ): double<br>+ GetFTemp( ): double |

| t : Temperature |
| --- |
| + ctemp : double |

图 1-9　一个温度类 Temperature 的类图及其对象 t 的对象图例子

正规的软件开发项目在程序设计结束后，应形成书面的程序设计报告，例如概要设计报告和详细设计报告。

## 1.3.2　程序实现

程序实现是使用计算机语言将程序设计阶段所得到的设计结果编写成计算机可以执行的程序。简单地说，程序实现就是编写程序的指令代码，简称**编码**（coding）。正式编码之前，程序员需要预先选择好编码所用的编程语言以及使用该语言编码所需的开发工具。

**编程语言**（programming language）是为了向计算机下达指令而专门设计的人工语言，其中定义了字符集、词法规则和语法规则。计算机语言可分为机器语言、汇编语言和高级语言三类。高级语言便于程序员学习和掌握，绝大部分程序员都使用高级语言。高级语言也有很多种，比较常用的有 Basic、C、C++、Java、C#、Python 等。表 1-1 给出了 2017 年 1 月的全球编程语言使用排行榜（TIOBE 指数），其中 C 语言位列排行榜的第 2 位，C++语言列第 3 位。

**表 1-1　2017 年 1 月全球高级语言使用排行榜（TIOBE 指数）**

| 2017 年 1 月排名 | 编程语言 | 使用率(%) | 2016 年 1 月排名 | 排名升降 |
| --- | --- | --- | --- | --- |
| 1 | Java | 17.278 | 1 | — |
| 2 | C | 9.349 | 2 | — |
| 3 | C++ | 6.301 | 3 | — |
| 4 | C# | 4.039 | 4 | — |
| 5 | Python | 3.465 | 5 | — |
| 6 | Visual Basic .NET | 2.960 | 7 | ↑ |
| 7 | JavaScript | 2.850 | 8 | ↑ |

续表

| 2017 年 1 月排名 | 编程语言 | 使用率(%) | 2016 年 1 月排名 | 排名升降 |
|---|---|---|---|---|
| 8 | Perl | 2.750 | 11 | ↑ |
| 9 | Assembly language | 2.701 | 9 | — |
| 10 | PHP | 2.564 | 6 | ↓ |
| 11 | Delphi/Object Pascal | 2.561 | 12 | ↑ |
| 12 | Ruby | 2.546 | 10 | ↓ |
| 13 | Go | 2.325 | 54 | ↑ |
| 14 | Swift | 1.932 | 14 | — |
| 15 | Visual Basic | 1.912 | 13 | ↓ |
| 16 | R | 1.787 | 19 | ↑ |
| 17 | Dart | 1.720 | 26 | ↑ |
| 18 | Objective-C | 1.617 | 18 | — |
| 19 | MATLAB | 1.578 | 15 | ↓ |
| 20 | PL/SQL | 1.539 | 20 | — |

说明：TIOBE 指数（www.tiobe.com）是一种表明编程语言流行趋势的指标，每月更新。这个排行榜只是反映了编程语言的热门程度，并不能说明编程语言的好坏。

支持结构化程序设计方法的高级语言称为结构化程序设计语言，例如 C 语言。支持面向对象程序设计方法的高级语言称为面向对象程序设计语言，例如 Java 语言、C#语言等。C++语言同时支持结构化程序设计和面向对象程序设计方法，因此它既是一种结构化程序设计语言，也是一种面向对象程序设计语言。

使用高级语言编写的程序称为**源程序**。源程序中的高级语言指令需要翻译成等效的机器语言指令才能被计算机硬件识别和执行。源程序的翻译执行有两种方式：一是**编译执行**，将源程序一次性翻译成等效的机器语言程序（称为**目标程序**），计算机执行目标程序；二是**解释执行**，计算机逐条翻译执行源程序中的指令，边翻译边执行。例如，C++是编译执行的高级语言，而 Python 是解释执行的高级语言。

程序员需要借助一些开发工具软件才能完成编程工作，例如编辑源程序需要用到**编辑器**软件，翻译源程序则需要用到**编译器**软件。**集成开发环境**（Integrated Development Environment，简称 IDE）就是这样一种开发工具软件，它由软件工具集和环境集成机制两部分组成。软件工具集可支持软件开发的相关过程、活动和任务，环境集成机制为工具集成和软件的开发、维护及管理等提供统一支持。常用的集成开发环境有 Microsoft 公司开发的 Visual C++ 6.0、Visual Studio 2008/2010 等，主要支持 C、C++和 C#语言的程序开发。另外还有 Eclipse，这是一个非常流行的开源集成开发环境。Eclipse 以插件的形式支持多种语言的开发，例如 Java、C、C++和 Python 等。初学者还可以使用 Dev C++作为自己练习 C++编程的集成开发环境。Dev C++也是一款开源软件，能通过互联网搜索并免费下载其安装包（50MB 左右）。本书使用 Visual C++ 6.0（参见附录 A）作为所有示例程序的集成开发环境。

使用 C++语言编写程序通常可细分为 4 步：编码、编译、连接和调试。

（1）**编码（coding）**：编写和输入用 C++语言编写的源程序代码。源程序文件的扩展名通常为.cpp。程序员在文本型编辑器中完成源程序的编码工作。

（2）**编译（compiling）**：将源程序代码翻译成等效的机器语言代码（即目标程序）。翻译前首先检查源程序中是否存在语法错误。如无语法错误则将其翻译成目标程序，否则提示错误信息。目标程序文件的扩展名通常为.obj。编译由一种称为编译器的程序完成，编译器也称为编译程序。

（3）**连接（link）**：将多个目标程序连接成一个整体，生成一个可以被计算机执行的程序文件。在 Windows 操作系统上，可执行程序文件的扩展名为.exe。连接由一种称作连接器的程序完成，连接器也称作连接程序。

（4）**调试（debug）**：通过编译、连接和试运行等方法来检查程序中可能存在的错误。程序中的语法错误可以使用编译器和连接器来检查，而语义错误则需要通过试运行来人工检查。如果发现错误，则需要修改源程序并重新编译、连接。集成开发环境一般会提供一些帮助调试的工具程序。

程序员需要熟练掌握计算机语言的知识，并能熟练运用相关的开发工具。

### 1.3.3 程序测试

正规的软件开发项目在程序实现阶段结束后，应对程序进行独立、系统、完整的测试。测试通常使用测试用例对程序进行黑盒测试，即选取一些测试数据作为程序的输入，运行程序，检查程序的输出结果是否正确。测试过程中所发现的问题应予以修改。程序开发过程中存在错误是难以避免的，只有经过严格的测试才能保证最终软件产品的质量。

测试贯穿整个程序开发过程，不同开发阶段有着不同的测试目标。例如：

- **单元测试**是对单个程序单元进行测试；
- **集成测试**是把已测试过的程序单元组装起来进行测试；
- **确认测试**是检查程序是否满足需求分析报告所规定的各种需求；
- **系统测试**是把已确认的软件纳入实际运行环境中进行测试；
- **回归测试**是在软件维护阶段为检查因代码升级、修改引入的新错误而进行的测试。

### 1.3.4 程序发布

正规软件开发项目所开发的程序需要经过**程序发布**环节，最终形成一个完整的软件产品，这样才能交付给用户使用。程序发布包括的工作主要有：

- 确定程序交付的形式和载体，例如光盘或网络下载；
- 将可执行程序打包，并编写相应的安装程序；
- 编写程序使用手册以及其他必须的文档。

程序及其相关的文档，合在一起被称为**软件**（software）。

我们学习 C++语言程序设计就是要学习程序设计的基本原理，掌握 C++语言的语法知识，并能熟练运用 C++语言编写计算机程序，从而完成从普通用户到程序员的角色转换。可以用以下几句话来描述程序与程序员、用户和计算机之间的关系。

- 程序是由程序员编写的；
- 一个程序可制成多个拷贝，分发给多个用户，由用户安装在各自的计算机上；
- 用户启动程序，由计算机来具体执行程序中的指令；
- 程序执行过程中可能需要用户输入原始数据或选择功能项，并查看输出结果，这就是用户与程序的交互；
- 程序员在编写程序过程中需考虑计算机能否执行，以及用户使用是否方便等因素。

### 本节习题

1．下列关于 C++语言的描述，哪个是错误的？（　　）

A．C++语言支持结构化程序设计方法　　B. C++语言支持面向对象程序设计方法

C．C++语言是编译执行的　　D．C++语言是解释执行的

2．用 C++语言编写的程序被称为（　　）。

A．源程序　　B．目标程序

C．可执行程序　　D．编译程序

3．计算机不能直接执行 C++源程序，必须经过下列哪项操作才能执行？（　　）

A．编译　　B．连接

C．调试　　D．先编译，再连接

4．下列哪种语言不支持面向对象程序设计方法？（　　）

A．C 语言　　B．C++语言

C．Java 语言　　D．C#语言

5．下列哪项内容不属于本书的学习范畴？（　　）

A．程序设计的基本原理　　B．C++语言的语法知识

C．组装计算机　　D．使用 C++语言编写程序

## 1.4　信息分类与数据类型

为便于使用，人们对客观世界中不同形式的信息进行了分类，例如文字、声音、图像等。但计算机只能处理数值形式的数据，任何信息都必须转成数值形式的数据（称为数字化）才能被计算机存储和处理。换句话说，任何信息只要能被数字化，就能被计算机处理。数字化后，任何信息处理问题都变成了数值计算问题。

人类社会使用十进制计数，并基于十进制进行数值运算。十进制是一种**计数制**，或简称为**数制**。为了便于硬件实现，计算机采用的是二进制。

### 1.4.1　二进制数制

一个 ***R* 进制**数制，采用 *R* 个基本计数符号，*R* 称为数制的**基**。运算时逢 *R* 进位。不同位置对应不同的值，该值是以 *R* 为底的幂，称为对应位置的**权**。

**十进制**（decimal）数制采用 10 个基本计数符号（即 0，1，…，9），10 称为十进制的基。运算时逢 10 进位，不同位置对应不同的权，例如个位的权为 $10^0$，十位的权为 $10^1$，百位的权为 $10^2$……。一个十进制数可以按权展开，例如：

$$82.625 = 8\times10^1 + 2\times10^0 + 6\times10^{-1} + 2\times10^{-2} + 5\times10^{-3}$$

### 1．二进制

**二进制**（binary）数制采用 2 个基本计数符号（即 0 和 1），2 称为二进制的基。运算时逢 2 进位，不同位置对应不同的权，例如 $2^0$，$2^1$，$2^2$，…。一个二进制数可以按权展开，例如：

$$1010010.101 = 1\times2^6+0\times2^5+1\times2^4+0\times2^3+0\times2^2+1\times2^1+0\times2^0+ 1\times2^{-1}+0\times2^{-2}+1\times2^{-3}$$

计算 1010010.101 右边所展开的多项式，得到结果 82.625。也就是说，二进制数 1010010.101 对应的十进制数为 82.625。计算按权展开多项式就可以实现二进制到十进制的转换。

将十进制数转换成二进制，需要将整数部分和小数部分分开，采用不同的方法分别进行转换。例如，将十进制数 82.625 转换成二进制，需要将整数部分 82 和小数部分 0.625 分开，分别进行转换。

十进制整数转二进制：**除 2 取余**。将十进制整数连续除以 2，取余数，直到商为 0。将所得余数按倒序排列即为转换结果。

| 除以 2 | 商(整数) | 余数 | | 二进制 |
|---|---|---|---|---|
| 82÷2 = | 41 | 0 | 低位 | |
| 41÷2 = | 20 | 1 | ↑ | |
| 20÷2 = | 10 | 0 | ↑ | |
| 10÷2 = | 5 | 0 | ↑ | 1010010 |
| 5÷2 = | 2 | 1 | ↑ | |
| 2÷2 = | 1 | 0 | ↑ | |
| 1÷2 = | 0 | 1 | 高位 | |

十进制小数转二进制：**乘 2 取整**。将十进制小数连续乘以 2，取整数，直到小数为 0，或达到所要求的精度位数（某些十进制小数不能精确转换成二进制）。将所得整数顺序排列即为转换结果。

| 乘以 2 | 乘积 | 取整 | | 二进制 |
|---|---|---|---|---|
| 0.625×2 = | 1.25 | 1 | 高位 | |
| 0.25×2 = | 0.5 | 0 | ↓ | 0.101 |
| 0.5×2 = | 1.0 | 1 | 低位 | |

十进制数 82.625 转换成二进制的结果是：1010010.101。可以看出，表示相同的数值，二进制需要更多的位数。

某些十进制小数转换成二进制时需要截断，例如：0.6。

| 乘以 2 | 乘积 | 取整 | | 二进制 |
|---|---|---|---|---|
| 0.6×2 = | 1.2 | 1 | 高位 | |
| 0.2×2 = | 0.4 | 0 | ↓ | |
| 0.4×2 = | 0.8 | 0 | | 0.1001 |
| 0.8×2 = | 1.6 | 1 | | |
| 0.6×2 = | 1.2 | 1 | | |
| … | … | … | 低位 | |

0.6 转换成二进制后是一个无限循环的有理数：0.10011001…。在计算机中存储二进制的有理数或无理数需要截断，只保留有限位数的小数。

### 2．八进制

在程序设计中还可能需要用到八进制和十六进制。**八进制**（octal）数制采用 8 个基本计数符号（即 0，1，…，7），8 称为八进制的基。运算时逢 8 进位，不同位置对应不同的权，例如 $8^0$，$8^1$，$8^2$，…。八进制与二进制可以按照表 1-2 的对应关系进行转换。1 位八进制对应 3 位二进制。

二进制转八进制的方法是：以小数点为基准，整数部分向左 3 位一组进行分组，不足 3 位高位补 0；小数部分向右 3 位一组进行分组，不足 3 位低位补 0，然后将每组的 3 位二进制转换为 1 位八进制，得到转换结果。例如，二进制数 1010010.101 转换为八进制的结果为 122.5。

1010010.101　**分组→**　001，010，010.101　**转换→**　122.5

八进制转二进制的方法就是上述过程的逆向过程，直接将每位八进制数转换为 3 位二进制数即可。

**表 1-2　八进制与二进制的对应关系**

| 八进制 | 二进制 | 八进制 | 二进制 |
|---|---|---|---|
| 0 | 000 | 4 | 100 |
| 1 | 001 | 5 | 101 |
| 2 | 010 | 6 | 110 |
| 3 | 011 | 7 | 111 |

### 3．十六进制

**十六进制**（hexadecimal）数制采用 16 个基本计数符号（即 0，1，…，9，再加上 6 个英文字母 A~F），16 称为十六进制的基。运算时逢 16 进位，不同位置对应不同的权，例如 $16^0$，$16^1$，$16^2$，…。十六进制与二进制可以按照表 1-3 的对应关系进行转换。1 位十六进制对应 4 位二进制。

表 1-3　十六进制与二进制的对应关系

| 十六进制 | 二进制 | 十六进制 | 二进制 |
|---|---|---|---|
| 0 | 0000 | 8 | 1000 |
| 1 | 0001 | 9 | 1001 |
| 2 | 0010 | A | 1010 |
| 3 | 0011 | B | 1011 |
| 4 | 0100 | C | 1100 |
| 5 | 0101 | D | 1101 |
| 6 | 0110 | E | 1110 |
| 7 | 0111 | F | 1111 |

十六进制与二进制之间的转换方法与八进制类似，区别就是 1 位十六进制对应 4 位二进制。例如，二进制数 1010010.101 转换为十六进制的结果为 52.A。

1010010.101　**分组→**　0101，0010 . 1010　**转换→**　52.A

## 1.4.2　数据类型

计算机如何在存储器中存储一个二进制数呢？这需要考虑两个方面的因素，分别是存储位数和存储格式。

### 1．存储位数

计算机管理存储器（含内存和外存）的最小单位是字节，每个字节可存储一个 8 位二进制数。因为位数的限制，1 个字节能存储的最大值为 $(11111111)_2$，即十进制的 $(255)_{10}$，其中下标 2 表示二进制，10 表示十进制。1 个字节能存储的最小值为 0，即 $(00000000)_2$。我们说，1 个字节所能存储的数值范围为 $(00000000)_2$~$(11111111)_2$，即十进制的 0~255。为了管理方便，计算机以固定的位数来存储二进制数，不足部分在高位补 0。这种使用固定位数存储数据的形式称为**定长存储**。可以将多个字节合在一起，这样可以增加存储位数，扩展可存储的数值范围。定长存储所采用的位数都是 8 的整数倍，例如 8 位（1 字节）、16 位（2 字节）或 32 位（4 字节）等，其对应的数值范围分别为 0~255、0~65 535、0~4 294 967 295。

程序所处理的数据只有存放到内存后才能被 CPU 处理，因此程序员应首先向计算机系统申请保存数据所需的内存空间。申请内存时，要指定存放数据所需的存储位数。**存储位数**越多，可存储的数值范围就越大，相应地所占用的内存空间也越大。因此，程序员在编写程序时应根据所处理数据可能的取值范围合理地选择存储位数。

### 2．存储格式

计算机存储二进制数还需要考虑的另外一个因素是存储格式。**存储格式**包括以下两个方面：以什么格式来区分正负数？以什么格式来区分整数和实数？

如果所处理的数据有正数，也有负数，计算机该如何存储一个数的正负号呢？假设存储位数为 8 位，可以将最高位拿出来作为符号位（0 表示正数，1 表示负数），剩余的 7 位用来存放数值，这种含符号位的存储格式称为**有符号**格式。相应地，不含符号位的存储格式则称为**无符号**格式。有符号格式可以存储正数，也可以存储负数。无符号格式只能存储非负数，即零或正数。

含符号位的二进制编码形式称为**原码**。例如 $(+82)_{10}$ 的原码是 $(\mathbf{0}\ 1010010)_2$，而 $(-82)_{10}$ 的原码是 $(\mathbf{1}\ 1010010)_2$。在采用有符号格式存储时，计算机使用原码的形式来存储正数，而存储负数时则使用另一种被称为**补码**的形式来存储。下面以 $(-82)_{10}$ 为例来演示负数补码的计算方法。

（1）$(-82)_{10}$ 的原码是 $(\mathbf{1}\ 1010010)_2$，其中最高位为符号位，1 表示负数；

（2）对数值部分求反，符号位不变，得到 $(\mathbf{1}\ 0101101)_2$，这种编码称为**反码**；

（3）将反码加 1，得到补码 $(\mathbf{1}\ 0101110)_2$。

补码与存储位数有关。存储位数不同，所转换出的补码是不同的。例如 $(-82)_{10}$ 的 16 位补码为：

$$(\mathbf{1}\ 0000000\ 01010010)_2 \rightarrow (\mathbf{1}\ 1111111\ 10101101)_2 \rightarrow (\mathbf{1}\ 1111111\ 10101110)_2$$

为统一起见，C++语言做出如下定义：正数的补码、反码与原码相同。这样我们可以说，在采用有符号格式存储时，计算机采用补码形式来存储数据（包括正数和负数）。计算机引入补码的原因有两个。

（1）定长存储时，“**A−B**”等于“A **的补码** + (−B)**的补码**”。这样可以将减法运算统一成加法运算，从而简化 CPU 的硬件设计；

（2）采用补码存储，0 的编码是唯一的。“+0”和“−0”的原码不同，具有二义性。如果采用 8 位存储，则它们的原码分别为 $(\mathbf{0}\ 0000000)_2$ 和 $(\mathbf{1}\ 0000000)_2$，而它们的补码都是 $(\mathbf{0}\ 0000000)_2$。

**存储格式**不同，其可存储的**数值范围**也不同（参见表 1-4）。设计程序时，程序员应根据所处理数据的特点合理地选择存储位数和存储格式。

**表 1-4　不同存储位数和存储格式情况下的数值范围**

| 存储位数　（字节数） | 数值范围 | |
|---|---|---|
| | 无符号格式 | 有符号格式 |
| 8(1) | 0~255 | –128~+127 |
| 16(2) | 0~65 535 | –32 768~+32 767 |
| 32(4) | 0~4 294 967 295 | –2 147 483 648~2 147 483 647 |

计算机如何存储一个实数呢？这里先介绍一下数的**科学表示法**。例如，一组实数：

82.625，8.262 5，0.826 25，0.082 625

它们的科学表示法分别为：

$$0.826\ 25\times10^2，0.826\ 25\times10^1，0.826\ 25\times10^0，0.826\ 25\times10^{-1}$$

一个 $R$ 进制数实数 $N$ 的**科学表示法**可以写成：

$$N = M\times R^E$$

其中，$E$ 是 $R$ 的指数，称为 $N$ 的**阶码**，阶码反映了小数点的位置。$M$ 表示 $N$ 的全部有效数字，称为 $N$ 的**尾码**（或称**尾数**），尾码反映了数据的精度。

计算机存储实数时，先将其转换成科学表示法，然后只存储其中的阶码和尾码。这种存储实数的格式称为**浮点格式**。下面以 $(-8.262\ 5)_{10}$ 为例来说明实数在计算机中的存储格式。

（1）将 $(-8.262\ 5)_{10}$ 转换成浮点形式 $(-0.826\ 25\times10^1)_{10}$；

（2）将阶码 $(+1)_{10}$ 转换成二进制 $(+1)_2$；

（3）将尾码 $(-0.826\ 25)_{10}$ 转换成二进制 $(-0.110\ 1001\ 1100)_2$（注：只保留 11 位精度）；

（4）存储阶码和尾码的二进制编码（注：不同计算机的存储格式可能不同）。

下面给出的演示例子用 4 位来存储阶码的补码，用 12 位来存储尾码的补码，共 16 位（占 2 个字节）。

| | | | |
|---|---|---|---|
| 0 | 001 | 1 | 001 0110 0100 |
| 阶码<br>符号位 | 阶码 | 尾码<br>符号位 | 尾码 |

### 3. 数据类型

总结一下，计算机存储二进制数据要考虑两个因素，即存储位数和存储格式，它们共同决定了可存储的数值范围。存储非负整数可以使用无符号格式；如需存储负数，则必须使用有符号格式；如需存储实数，则必须使用浮点格式，即“阶码+尾码”的存储格式。因为计算机使用定长存储，如果程序员选择不当，则保存数据时可能出现溢出或损失精度等问题。

为了让程序员在申请内存时能方便地指定存储位数和存储格式，计算机高级语言引入了数据类型的概念。结合实际应用的需要，高级语言一般都预定义了若干种数据类型，规定了每种数据类型的存储位数、有无符号位、存储整数或实数等。下面列出 C++语言中的几种数据类型实例。

- int：存储位数为 32 位（4 字节），有符号位，可存储整数。
- unsigned int：存储位数为 32 位（4 字节），无符号位，只能存储非负整数。
- double：存储位数为 64 位（8 字节），有符号位，以浮点格式存储实数。

**数据类型**（type）规定了数据的存储位数和存储格式。程序员在申请内存空间时应根据所存储数据可能的取值范围合理地选择数据类型，该数据类型决定了所申请内存空间的字节数及其存储格式。C++语言将预定义的数据类型称为**基本数据类型**。

## 1.4.3 信息分类及数字化

人们需要通过输入设备将客观世界中不同形式的信息**数字化**（digitization）成数值形式，然后才能输入到计算机进行存储和处理。例如，通过键盘输入文字，通过麦克风输入声音，通过图像扫描仪或数码相机输入图像等，这些输入过程都是一种数字化过程。

### 1．文字信息的数字化

计算机将阿拉伯数字、英文字母以及一些常用符号等字符以按键的形式制成键盘。键盘能感知用户的按键动作，并将按键所对应的字符转换成一个数值代码。例如，用户单击一次按键 A，键盘就向计算机输入了一个数值 65，65 是字符 A 的数值代码。计算机使用统一的标准来对字符进行编码，这个标准是由美国国家标准学会 ANSI 制定的美国信息交换标准代码（American Standard Code for Information Interchange），简称 ASCII 码。ASCII 码表（表 1-5）包括 10 个阿拉伯数字、52 个英文字母（含大小写）以及 33 个其他常用符号，另外还包括 33 个控制字符（它们是不可见字符或不可打印字符，例如 Esc 是一个控制字符，其代码为 27），合计 128 个字符。

表 1-5　ASCII 码表

| ASCII 值 | 字符 | ASCII 值 | 字符 | ASCII 值 | 字符 | ASCII 值 | 字符 |
|---|---|---|---|---|---|---|---|
| 0 | NUT | 32 | (space) | 64 | @ | 96 | 、 |
| 1 | SOH | 33 | ! | 65 | A | 97 | a |
| 2 | STX | 34 | ” | 66 | B | 98 | b |
| 3 | ETX | 35 | # | 67 | C | 99 | c |
| 4 | EOT | 36 | $ | 68 | D | 100 | d |
| 5 | ENQ | 37 | % | 69 | E | 101 | e |
| 6 | ACK | 38 | & | 70 | F | 102 | f |
| 7 | BEL | 39 | , | 71 | G | 103 | g |
| 8 | BS | 40 | ( | 72 | H | 104 | h |
| 9 | HT | 41 | ) | 73 | I | 105 | i |
| 10 | LF | 42 | * | 74 | J | 106 | j |
| 11 | VT | 43 | + | 75 | K | 107 | k |
| 12 | FF | 44 | , | 76 | L | 108 | l |
| 13 | CR | 45 | - | 77 | M | 109 | m |
| 14 | SO | 46 | - | 78 | N | 110 | n |
| 15 | SI | 47 | / | 79 | O | 111 | o |
| 16 | DLE | 48 | 0 | 80 | P | 112 | p |
| 17 | DCI | 49 | 1 | 81 | Q | 113 | q |
| 18 | DC2 | 50 | 2 | 82 | R | 114 | r |
| 19 | DC3 | 51 | 3 | 83 | X | 115 | s |
| 20 | DC4 | 52 | 4 | 84 | T | 116 | t |
| 21 | NAK | 53 | 5 | 85 | U | 117 | u |
| 22 | SYN | 54 | 6 | 86 | V | 118 | v |
| 23 | TB | 55 | 7 | 87 | W | 119 | w |
| 24 | CAN | 56 | 8 | 88 | X | 120 | x |
| 25 | EM | 57 | 9 | 89 | Y | 121 | y |
| 26 | SUB | 58 | : | 90 | Z | 122 | z |
| 27 | ESC | 59 | ; | 91 | [ | 123 | { |
| 28 | FS | 60 | < | 92 | \ | 124 | \| |
| 29 | GS | 61 | = | 93 | ] | 125 | } |
| 30 | RS | 62 | > | 94 | ^ | 126 | ~ |
| 31 | US | 63 | ? | 95 | _ | 127 | DEL |

ASCII 码表中编码的数值范围是 0~127，可以用 7 位二进制来表示，即 000 0000~111 1111。计算机一般是用一个字节（8 位）来存储一个字符编码，最高位未用到，置为 0。

中文字符则使用 1980 年由中国国家标准总局发布《信息交换用汉字编码字符集》中所确定的编码标准，简称 GB 2312 标准。该标准共收入 6 763 个常用汉字和 682 个图形字符。整个字符集分成 94 个区，每区有 94 个位，每个区位对应一个字符。根据所在的区和位进行编码，得到**区位码**。将区位码换算成十六进制再加上十六进制的（2 020）$_{16}$，得到**国标码**。国标码再加上（8 080）$_{16}$，就得到最终存储在计算机里的汉字编码，即**机内码**。存储一个汉字的机内码需要 2 个字节，每个字节的最高位都为 1。可以通过字符编码的最高位来区分中英文字符，0 对应英文字符，1 对应中文字符。中文输入法负责将用户输入的中文字符转换成数值编码。目前还有一些新的汉字编码标准，例如《汉字编码扩展规范》（简称 GBK 标准）、Unicode 编码等。

## 2．声音信息的数字化

计算机通过麦克风输入声音信息。麦克风将声音的振动信号转换成电信号，由一些相关的电路按某种固定时间间隔（例如 0.02ms）对电信号进行离散化采样，然后再按某种固定的位数（例如 8 位二进制）将采样得到的模拟电信号转换为数字信号（称为 A/D 转换），最终一段声音信号被转换成一组二进制数值。数值的大小对应声音信号中振幅的大小。假设使用 8 位 A/D 转换，则每个数值可以用 8 位二进制存储，即 1 个字节。

通过数字化，计算机可以对声音信息进行存储和处理。存储声音信息就是存储一组数值。声音信息的处理被转换成对这组数值的某种运算，例如调整这组数值的大小等价于调整音量的大小。处理后的声音数据可按照其数字化的逆过程通过输出设备（例如音箱）重新还原成声音信号。

## 3．图像信息的数字化

计算机通过图像扫描仪或数码相机输入图像信息。图像扫描仪或数码相机中的光电转换器件（例如 CCD）将光信号转换成 3 路电信号（分别对应红、绿、蓝 3 基色），由一些相关的电路按某种固定间距对电信号进行离散化采样（例如 200DPI 扫描分辨率表示每英寸 200 个采样点），然后再按某种固定的位数将采样得到的模拟电信号转换为数字信号，最终一幅图像信号被转换成一组二进制数值（类似于一个矩阵）。图像中的每个采样点称为一个像素。数值的大小对应图像信号中光强的强弱。假设使用 8 位 A/D 转换，则每个像素可以用 3 个 8 位二进制存储，即 3 个字节，其数值分别对应像素的红、绿、蓝光强。

通过数字化，计算机可以对图像信息进行存储和处理。存储图像信息就是存储一组数值。图像信息的处理被转换成对这组数值的某种运算，例如调整这组数值的大小等价于调整图像的亮度。处理后的图像数据可按照其数字化的逆过程通过输出设备（例如显示器）重新还原成图像信号。

## 本节习题

1．十进制 19 转换成二进制后的结果为（　　）。
A．10001　　B．10010　　C．10011　　D．10100
2．十进制 19 转换成八进制后的结果为（　　）。
A．21　　B．22　　C．23　　D．24
3．十进制 19 转换成十六进制后的结果为（　　）。
A．11　　B．12　　C．13　　D．14
4．十进制 19.625 转换成二进制后的结果为（　　）。
A．10001.101　　B．10010.011　　C．10011.101　　D．10100.011
5．程序设计中的数据类型与下列哪个概念没有关联？（　　）
A．存储位数　　B．存储格式　　C．取值范围　　D．数据来源
6．计算机是以（　　）的形式来存储实数的。
A．原码　　B．反码　　C．补码　　D．阶码＋尾码
7．目前还有哪些信息不能被数字化？（　　）
A．文字信息　　B．听觉信息　　C．视觉信息　　D．味觉信息

# 1.5 C++语言简介

高级语言是在不同程序设计思想指导下所设计的人工语言。1972 年，由 Dennis Ritchie 创建的 C 语言是一种结构化程序设计语言，具有简洁的语法和优异的性能。1989 年，美国国家标准学会（ANSI）正式发布 C 语言标准，简称 ANSI C 或 C89。1990 年，该标准被国际标准化组织（ISO）采纳（仅有一些小的修改），简称 ISO C 或 C90。因为有了国际标准，计算机语言方成为国际上真正统一的语言，也就是说全世界程序员使用的 C 语言都是相同的。C 语言是 20 世纪 80~90 年代使用最为广泛的计算机语言，并一直沿用到今天。C 语言将结构化程序设计思想推向了顶峰。

在面向对象程序设计思想出现以后，人们开始设计各种支持面向对象程序设计的计算机语言。C 语言具有广泛的程序员基础，因此在 C 语言基础上发展新的支持面向对象程序设计的计算机语言成为一种必然的选择。于是，1983 年 C++诞生了。C++语言兼容 C 语言的所有语法功能，是 C 语言的超集。C++语言扩展了“类”等面向对象的语法形式，因此也被称为“带类的 C”。1998 年，国际标准化组织（ISO）正式发布 C++语言标准，简称 C++98。虽然之后又陆续发布了 C++03、C++11 和 C++14 等新标准，但 C++98 标准仍为 C++语言最基础、最通用的标准。C++语言支持面向对象程序设计，同时也支持传统的结构化程序设计。其知识结构全面，知识点从结构化程序设计逐步过渡到面向对象程序设计，是学习程序设计的首选入门语言。

1995 年，美国 SUN 公司推出了 Java 语言。Java 语言是针对跨平台（即 Java 程序无需重新编译即可在不同操作系统上运行）和互联网应用程序而设计的计算机语言。因为近些

年来互联网的蓬勃发展，Java 语言一跃而成为全球高级语言使用排行榜的第 1 名。Java 语言是“纯”面向对象的程序设计语言，不支持结构化程序设计。和 C++语言相比，Java 语言对 C 语言语法进行了精简，或者说只是部分“借鉴”了 C 语言的语法特点，因而语法相对简单，易于上手。但 Java 语言放弃了 C 语言中一些与硬件相关的重要概念，例如“指针”，因此在对程序设计基本原理和概念的理解上存在欠缺。反过来，掌握了 C++语言，Java 语言的学习就会比较简单，甚至可以自学。

学习程序设计与学习某一种计算机语言不是同一个概念。程序设计与计算机语言的关系就如同人的“思想”与“语言”的关系。程序设计是思想，是解决问题的方法；计算机语言是用来表达思想的形式，是描述问题及其解决方法的指令，因此指令在计算机语言中也被称作**语句**（statement）。作为初学者，重要的是学习程序设计的原理和方法，培养运用计算思维来分析和解决问题的能力。同时，还需要选择一种计算机语言作为入门语言，学习其语法规则，并能熟练地阅读和编写一些简单的计算机程序。计算机语言将始终贯穿于程序设计的学习过程之中。程序设计的原理和方法是共同的，而计算机语言可以是不同的。一旦掌握了程序设计原理和某种计算机语言，其他的语言可以通过一些进阶培训，甚至是自学就可以完成。本书选择 C++语言作为学习程序设计的入门语言。

### 学习本章的要点

- 读者要从有形的硬件来理解相对抽象的软件。
- 读者要认识到，计算机中的数据是有类型的。类型决定了数据在计算机中的存储位数和存储格式。
- 读者要知道，学习程序设计和学习编程语言不是一回事。和 C 语言、Java 语言相比，C++语言的知识体系更加系统全面。本书选用 C++语言作为程序设计初学者的入门语言。

## 1.6 本章习题

**1．模仿编程**。例 1-1 所示的 C++程序能够将摄氏温度换算成华氏温度。请模仿编写一个将华氏温度换算成摄氏温度的 C++程序。使用 C++语言集成开发环境（例如 Visual C++ 6.0 或 Dev C++等）组建并执行该程序。记录编程过程中所出现的问题及解决方法。

注：C++语言用“/”表示除法。

**2．模仿编程**。模仿例 1-1 所示的 C++程序，编写一个计算圆面积的程序。使用 C++语言集成开发环境组建并执行该程序。记录编程过程中所出现的问题及解决方法。

# 第2章 数值计算

**数值计算**（numerical computation）就是利用计算机求解各种数学问题。

例如，将摄氏温度换算成华氏温度的公式是：$f = c×1.8 + 32$，其中$f$表示华氏温度，$c$表示摄氏温度。我们可以手工完成温度的换算。换算时，先指定变量$c$的数值（即待换算的摄氏温度值），然后按照公式计算得到变量$f$的数值（即对应的华氏温度值）。

如果想让计算机来帮我们做温度换算的工作，就需要编写一个数值计算程序。运行该程序，输入一个摄氏温度，程序就能自动换算成华氏温度，并将结果显示出来。编写这样的温度换算程序，我们需考虑以下4个方面的问题。

**（1）数据存储**。CPU 只能处理存放在内存中的数据，因此温度换算程序首先要申请内存空间来存放摄氏温度和华氏温度这两个数据。**C++语言通过定义变量来申请内存空间**。每定义一个变量，计算机即为该变量分配内存空间。定义变量后，可以向该变量所分配的内存单元写入数据或读出其中的数据。

在计算机语言中，一条指令称为一条语句。C++语言有多种不同功能的语句，每种语句都有自己的语法规则。使用定义变量语句，温度换算程序可以定义两个变量 ctemp 和 ftemp 来分别保存摄氏温度和华氏温度。变量名可以直接用原始公式中的$c$和$f$，但 ctemp、ftemp 更便于程序员理解和记忆（其中 temp 是温度 temperature 的缩写）。

**（2）数据输入**。温度换算程序应让使用该程序的用户能够输入想要换算的摄氏温度。C++语言使用输入语句来接收用户输入的数据，并保存到预先定义好的变量 ctemp 中。

**（3）数值计算**。温度换算程序将存放在变量 ctemp 中的摄氏温度按照公式换算成华氏温度，换算结果要保存到华氏温度对应的变量 ftemp 中。C++语言以表达式的形式来编写换算公式。单个表达式仅能描述一个简单的计算。复杂计算可能需要分多步完成，每一步会产生一个中间结果，计算机程序也要预先定义好保存这些中间结果的变量。

**（4）数据输出**。温度换算程序要把计算所得到的结果反馈给用户，例如将保存在变量 ftemp 中的华氏温度换算结果输出到显示器上。C++语言使用输出语句来显示保存在某个变量中的数据。

## 2.1 程序中的变量

数据是程序处理的对象。数据要存放在内存中才能被 CPU 读取和处理，处理后的结果也只能保存回内存中。程序中的数据包括原始数据、中间结果、最终结果等，C++语言使用**变量**（variable）来保存这些数据。**定义变量**就是为变量申请内存空间。定义变量后，可

以向该变量所分配的内存单元写入数据或读出其中的数据，这称为**访问变量**。程序执行时，程序中的变量就对应内存中的某个内存单元。程序结束退出时，变量将释放其所占用的内存单元，以便给其他程序继续使用。简单地说，程序中的**变量＝内存单元**。

## 2.1.1 变量的定义

程序员在定义变量时要考虑三方面的内容：变量如何在内存中存储，变量如何命名，以及按照语法规则编写定义变量语句。

### 1. 变量如何在内存中存储

不同类型数据有不同的数值范围，所需要的存储位数不一样。例如，月份可以用整数表示，其数值范围为 1~12，转换为二进制为（1）$_2$~（1100）$_2$。以二进制形式来存储月份数据需要 4 位。内存是以字节（8 位）为单位来管理的，程序员可以申请 1 个字节来存储月份数据，不足 8 位时高位补 0。而对于摄氏温度这样的数据，其数值可能为负数，也可能为实数。计算机存储负数需要用有符号格式（即补码格式），存储实数需要用阶码+尾码的格式（即浮点格式），这些格式统称为存储格式。程序员定义变量时，要根据数据可能的取值范围指定变量的**存储位数**和**存储格式**。

为了让程序员定义变量时能方便地指定存储位数和存储格式，计算机高级语言引入了**数据类型**的概念。在 C++语言中，所有变量都是有类型的。程序员在定义变量时需指定其数据类型（也就是指定其存储位数和存储格式）。C++语言预先定义了若干种**基本数据类型**，可满足绝大部分数值计算问题的需要。C++语言规定了每种基本数据类型的存储格式（例如有无符号位、存储整数或实数），而存储位数则会随操作系统或编译器版本的不同而有所不同。表 2-1 列出了 32 位 Windows 操作系统上 10 种数值计算常用的基本数据类型。

**表 2-1 C++语言中 10 种数值计算常用的基本数据类型（32 位 Windows 操作系统）**

| 数据类型 | 说明 | 存储位数（字节数） | 可存储的数值范围 | 运算 |
|---|---|---|---|---|
| char 或 signed char | 有符号字符型 | 8 位（1） | –128~127 | 算术运算<br>关系运算 |
| unsigned char | 无符号字符型 | | 0~255 | |
| short 或 signed short | 有符号短整型 | 16 位（2） | –32 768 ~ 32 767 | |
| unsigned short | 无符号短整型 | | 0 ~ 65 535 | |
| int 或 signed int | 有符号整型 | 32 位（4） | –2 147 483 648 ~ 2 147 483 647 | |
| long 或 signed long | 有符号长整型 | | | |
| unsigned 或 unsigned int | 无符号整型 | 32 位（4） | 0 ~ 4 294 967 295 | |
| unsigned long | 无符号长整型 | | | |
| float | 单精度浮点型 | 32 位（4） | $3.4\times10^{-38}$ ~ $3.4\times10^{38}$（绝对值精度） | |
| double | 双精度浮点型 | 64 位（8） | $1.7\times10^{-308}$ ~ $1.7\times10^{308}$（绝对值精度） | |

表 2-1 中，char、short、int 和 long 都可以存放整数（统称为**整型**），其区别是所占用内存的字节数不同。字节数越大，可以存储的数值范围就越大，但需占用更多的内存。如果超出数据类型的数值范围，则会造成数据丢失，这称为**数据溢出**（data overflow）。存储非负整数可以使用**无符号类型**（unsigned）。存储实数应使用 float 或 double 类型（统称为**浮点型**）。double 类型的精度更高（可保留更多的小数位数），数值范围也更大。程序员应根据所处理数据可能的取值范围来判断应定义哪种类型的变量，所依据的原则是：既要保证精度、防止溢出，又要尽可能少地占用内存。

例如在温度换算公式 $f = c\times1.8 + 32$ 中，摄氏温度 $c$ 是自变量，其取值范围称为定义域。华氏温度 $f$ 是因变量，其取值范围称为值域。在实际生活中，温度数据通常是实数，因此定义保存温度数据的变量应选择 float 或 double 类型。float 类型已经能够满足温度换算程序的精度要求（可以保留 38 位小数），其所占用内存的字节数为 4 字节，是 double 类型的一半，因此选用 float 类型更加合理。

不同类型的数据可以做不同的运算。例如在 C++语言中，整数、实数都可以进行算术运算（即加减乘除），也可以进行关系运算（即比较大小），但整数还可以进行一类被称为“位运算”的运算，而实数不可以。因此在计算机语言中，数据类型的内涵除了包括数据的存储位数和存储格式之外，还隐含包括了该类型数据可以进行哪些运算。

### 2．如何为变量命名

程序员在定义变量时除了指定变量的数据类型外，还需要指定变量名，即为**变量命名**。C++语言的词法元素包括关键字、标识符、常量、运算符、分隔符等。**关键字**（keyword）是 C++语言预先保留的具有特定含义的单词。例如，表 2-1 中表示基本数据类型所用到的单词 int、float、unsigned 等，它们就是 C++语言的关键字。表 2-2 列出了 63 个 C++语言所保留的关键字。

**表 2-2　C++语言中的 63 个关键字**

| | | | | |
|---|---|---|---|---|
| asm | do | if | return | typedef |
| auto | double | inline | short | typeid |
| bool | dynamic_cast | int | signed | typename |
| break | else | long | sizeof | union |
| case | enum | mutable | static | unsigned |
| catch | explicit | namespace | static_cast | using |
| char | export | new | struct | virtual |
| class | extern | operator | switch | void |
| const | false | private | template | volatile |
| const_cast | float | protected | this | wchar_t |
| continue | for | public | throw | while |
| default | friend | register | true | |
| delete | goto | reinterpret_cast | try | |

程序中所包含的一些实体（例如变量），需要程序员为它们命名。由程序员定义的程序实体名称统称为**标识符**（identifier）。对标识符的命名需符合如下命名规则：

- 以大写或小写英文字母、下画线“_”开头；
- 由大写或小写英文字母、下画线“_”、数字 0~9 组成；
- 不能是关键字。

例如下面这些例子：

abc、Abc、_bc、abc123、abc_123、A、a、_No1 等，**符合**标识符命名规则。

123、abc.123、温度、float 等，**不符合**标识符命名规则，属于语法错误。

另外，C++语言区分大小写英文字母。例如，abc 和 Abc 是两个不同的标识符。

#### 3．变量定义语句的语法规则

在计算机语言中，**语句**是一条语法完整的指令。C++程序中的语句应符合 C++语言的语法规则，并以分号“;”结束。请注意：结束符“;”是英文输入状态下输入的分号，不是中文输入状态下的分号，这是初学者易犯的一个错误。类似的还有逗号“,”、单引号“’”、双引号“” ”等分隔符。定义变量需要编写**变量定义**（variable definition）语句。

C++语法：变量定义语句

| 数据类型　变量名 1, 变量名 2, …, 变量名 n ; |
|---|
| 语法说明：<br>■ **数据类型**指定了变量的存储位数和存储格式。<br>■ **变量名**需符合标识符的命名规则。<br>■ 可在一条语句中定义多个具有相同数据类型的变量，变量之间用“,”隔开。 |
| 举例：定义 2 个变量 ctemp 和 ftemp<br>double ctemp ;　　// 计算机将在内存中为 double 型变量 ctemp 分配 8 个连续的字节<br>　　　　　　　　　// 作为其内存单元，并在该内存单元中以浮点格式存储数据<br>double ftemp ;<br>或<br>double ctemp, ftemp ;　// 可以在一条语句中定义多个相同类型的变量 |

程序由计算机执行。当执行到程序中的变量定义语句时，计算机为所定义的变量分配内存单元，后续语句将通过变量名来访问该内存单元。计算机只能识别机器语言，机器语言是通过地址访问内存的。高级语言则通过变量名来访问内存，变量名便于程序员记忆和使用。高级语言程序需编译成机器语言程序才能被计算机执行。编译时，程序中的变量名被转换成了内存地址。也就是说，程序员在编写高级语言程序时使用变量名来申请和访问内存，而计算机执行其编译后的机器语言程序时使用的则是内存地址。

### 2.1.2　变量的访问

定义变量后，可以向变量所分配的内存单元**写入**（write）数据或**读出**（read）其中的数据。对变量的读写操作统称为对变量的**访问**（access）。

C++语言对变量写入数据的操作有 3 种方式。

（1）使用**输入**语句，将键盘输入数据写入变量的内存单元。例如，

```
cin >> ctemp;
```

该语句指示计算机从键盘接收用户输入的数据，并将其写入变量 ctemp 的内存单元。

（2）使用**赋值**运算符“=”，对变量进行赋值运算。例如，

```
ctemp = 36;
```

该语句将数值 36 赋值给 ctemp，即将数值 36 写入变量 ctemp 的内存单元。

（3）**初始化**。定义变量的同时为变量赋初始值，这就是初始化。例如，

```
int x=10, y;
```

该语句定义了 2 个 int 型变量 x 和 y。执行该语句时，计算机为变量分配内存空间。x 被初始化了，计算机在为 x 分配内存单元的同时向该内存单元写入初始值 10。y 没有被初始化，通常情况下其所分配内存单元中的值是以前程序遗留下来的，是不确定的。

C++语言从变量读出数据的操作有 2 种方式。

（1）当变量作为**操作数**参与运算时，计算机将自动读取其内存单元中存放的数据。例如，

```
ftemp = ctemp*1.8 + 32;
```

该语句中等号右边的变量 ctemp 是作为操作数参与运算的，计算机会自动读取其内存单元中的数据。读出该数据后，再使用该数据进行运算，并将运算所得到的结果赋值给等号左边的变量 ftemp。

（2）使用**输出**语句，读出并显示变量内存单元中存放的数据，以便用户查看。例如，

```
cout << ftemp;
```

该语句指示计算机读出变量 ftemp 内存单元中存放的数据，并在显示器上显示出来。

定义后的变量才有内存单元，才能被访问。程序员在编写 C++程序时应遵循“**先定义，后访问**”的原则。未经定义的变量不能访问。

## 本节习题

1．每周有 7 天，为星期一~星期日分别赋予一个整数编码。使用十进制只需 1 位编码就够了，例如 0~6。请问用二进制最少需要几位编码？（　　）

A．1　　B．2　　C．3　　D．4

2．下列哪种数据类型占用内存的字节数最多？（　　）

A．char　　B．int　　C．float　　D．double

3．下列哪种数据类型的变量不能存储负数？（　　）

A．unsigned short　　B．int　　C．float　　D．double

4．计算机（32 位系统）存储 int 型数据使用下列哪种方式？（ ）

A．占用 2 字节，原码形式 B．占用 2 字节，补码形式

C．占用 4 字节，原码形式 D．占用 4 字节，补码形式

5. 假设变量 x 的值域为[0, 50 000]之间的整数，则其最适合的数据类型是哪种？（ ）

A．unsigned short B．int C．float D．double

6. 假设变量 x 的值域为[–1. 0, 1. 0]之间的实数，则其最适合的数据类型是哪种？（ ）

A．char B．short C．int D．double

7．下列哪个名字可以作为变量名？（ ）

A．No.1 B．123ABC C．long D．Long

8．下列定义变量语句中，错误的是（ ）。

A．int x, y; B．int x = 5, y;

C．int x = 5, y = 5; D．int x = y = 5;

## 2.2 程序中的常量

温度换算公式 $f = c \times 1.8 + 32$ 中，$f$ 和 $c$ 是变量，而 1.8 和 32 是常量。程序运行过程中数值不会改变的量称为**常量**（constant）。本节我们先介绍两种程序中常用的常量：字面常量和符号常量。

### 1．字面常量

C++语言借鉴了我们数学中熟悉的书写形式来表示数值常量。例如：

```
ftemp = ctemp*1.8 + 32;
```

该语句中，32 是一个整数常量，1.8 是一个实数常量，它们就是源程序中的字面常量（literal constant）。C++语言使用负号“–”来表示负数常量，例如–32、–1.8 等。实数常量也可以采用科学表示法，例如 1.8 可以写成 1.8e0、1.8E0、0.18e1、18.0e–1 等，其中表示指数的字母写成小写 e 或大写 E 都可以。

程序运行时，常量也需要存储在内存中。程序员在编写程序时应指定常量的数据类型，以确定该常量的存储位数和存储格式。C++语言采用默认形式或后缀形式来指定常量的数据类型。

**1）默认形式**

C++程序中，整数常量默认为 int 型，即有符号整型，存储位数 32 位（4 字节）。例如 10、–10、0 等，都默认为 int 型常量。

C++程序中，实数常量（带小数点）默认为 double 型（即双精度浮点型），存储位数 64 位（8 字节）。例如 10.5、–10.5、1.05e1、10.0 等，都默认为 double 型常量。

小数点是区分整数和实数的标志。例如 C++程序中，虽然 10 和 10.0 的数值相等，但数据类型是不一样的。10 是 int 型，以 4 字节补码形式存储；而 10.0 是 double 型，以 8 字节浮点形式存储。

**2）后缀形式**

可以在常量后面添加不同的后缀字母来指定常量的数据类型（表 2-3）。

**表 2-3　C++语言指定常量数据类型的后缀字母表**

| 后缀字母 | 数据类型 | 举　例 |
|---|---|---|
| 在整数常量后面添加 L 或 l | 长整型 long | 10L，10l，–20L |
| 在整数常量后面添加 U 或 u | 无符号格式 unsigned | 10U，10u，20UL，20LU |
| 在数值常量后面添加 F 或 f | 单精度浮点型 float | 10.5f，10.5F，20F |

C++程序中的数值常量默认为十进制数，这符合人的使用习惯。某些情况下，程序员可能需要在程序中以八进制或十六进制来表示整数常量。

以 0 开头的整数常量为八进制。例如 C++程序中的整数常量 020 是一个八进制数，其十进制数值等于 16。八进制整数常量中只能出现 0~7 的数字，否则就属于语法错误。例如，C++程序中不能出现 019 这样的整形常量。

以 0x 开头的整数常量为十六进制，其中可以包含 0~9 的数字和 A~F 的字母（大小写均可）。例如，C++程序中的整数常量 0x1a 是一个十六进制数，其十进制数值等于 26。

编写 C++程序时，程序员可以使用十进制、八进制或十六进制来表示数值常量。在编译成机器语言时，由编译器负责将不同进制的数值常量统统转换成二进制。

## 2．符号常量

可以将经常使用的常量定义成一个**符号常量**，或称为宏定义（macro definition），然后在程序代码中用符号常量来代替具体的数值。

**例 2-1　符号常量应用举例：计算圆的面积和周长**

```
//  C++程序实例：从键盘输入圆的半径，计算并显示圆的面积和周长
#include <iostream>
using namespace std;
#define   PI   3.14                // 定义一个符号常量 PI 来表示 π 的值

int main( )
{
    double r;                      // 定义一个变量 r 来存放圆的半径
    cin >> r;                      // 从键盘输入圆的半径

    double s;                      // 定义一个变量 s 来存放圆的面积
    s = PI * r * r;                // 计算圆的面积，结果保存到变量 s 中
    cout << s << endl;             // 显示圆面积
                // endl 表示在显示完面积后换一行，这样就能与下面将显示的周长隔开

    double len;                    // 定义一个变量 len 来存放圆的周长
    len = 2 * PI * r;              // 计算圆的周长，结果保存到变量 len 中
    cout << len << endl;           // 显示圆的周长
    return 0;
}
```

代码第 4 行定义一个符号常量 PI 来表示 π 的值，然后在第 12 和 17 行中用符号常量 PI 来代替具体的数值 3.14。使用符号常量有以下 3 个优点：

（1）**保证数值常量的一致性**。使用符号常量 PI 可以防止程序中出现不同的 π 值。例如，出现 3.14 或 3.141 592 6 等不同的 π 值。

（2）**提高程序代码的可读性**。例如，如果不使用符号常量，代码第 12 行原来的形式是：

```
s = 3.14 * r * r;
```

PI 是符号常量的名字，它明确地表示自己是一个 π 值。这样可以让阅读者更容易理解，更容易联想到求圆面积的公式。

（3）**便于数值常量的修改**。假设为了提高精度，需将源程序中的 3.14 统一改为 3.141 592 6。使用符号常量只要修改其定义语句，即代码第 4 行：

```
#define   PI   3.1415926
```

程序其他部分无须修改（例如代码第 12 和 17 行）。而若直接书写 3.14，则需要搜索整个源程序，逐一修改所有的 3.14。

**C++语法：定义符号常量**

| **#define 符号常量名 常量值** |
|---|
| 语法说明：<br>■ 以井号“#”开头，结尾不能加分号“;”。<br>■ 符号常量名应符合标识符命名规则。习惯上符号常量名使用大写字母。<br>■ 定义符号常量也称为宏定义，将在第 6.2 节进行详细讲解。 |
| 举例：定义一个符号常量 ABC<br>#define   ABC   5                    // 定义符号常量 ABC，来表示常量值 5 |

### 3．理解常量与变量

这里我们仍以温度换算公式 $f = c×1.8 + 32$ 为例，回顾一下什么是程序中的常量和变量。

程序设计中，常量就是在编写程序时就能确定其数值大小的量（例如 1.8 和 32），程序员在程序中直接书写数值（即字面常量），或将其定义成符号常量。而变量则是在编写程序时不能确定其数值大小的量（例如摄氏温度 $c$ 是今后程序执行时由用户输入的），程序员需要在编写程序时使用定义变量语句预先为它们分配好内存空间（例如为摄氏温度定义一个变量 ctemp），这样程序执行时才能在其中存放数据。

程序中的常量是在程序编写时由程序员指定数值大小，并且在程序执行过程中不会改变的量。程序中的变量是在程序编写时申请、执行时分配的内存单元，用于保存用户输入的数据或计算得到的结果。程序执行过程中变量是可变的，其含义是随着程序的执行，变量的内存单元中可能被存放不同的数值。

### 本节习题

1．C++源程序中，下列哪个常量的数据类型是 float 型？（　　）

A．10　　B．10L　　C．10.0　　D．10.0f

2．C++源程序中，下列哪个整数的数值最小？（　　）

A．15　　B．15L　　C．015　　D．0x15

3．下列哪个浮点型常量是错误的？（　　）

A．5.82　　B．0.582e1　　C．0.582E1f　　D．0x5.82

4．C++源程序中，数值常量 010 被默认为（　　）。

A．二进制，short 类型　　B．二进制，int 类型

C．八进制，int 类型　　D．十六进制，short 类型

5．下列符号常量定义语句中，正确的是（　　）。

A．#define _ABC 10　　B．define ABC 10

C．#define _ABC 10;　　D．#define ABC=10

6．计算圆形周长的公式是：周长=$2\pi r$，其中 $r$ 为半径。编写计算圆形周长的程序时需要将什么数据定义成变量？（　　）

A．π　　B．半径　　C．周长　　D．半径和周长

7．计算圆形周长的公式是：周长=$2\pi r$，其中 $r$ 为半径。编写计算圆形周长的程序时需要将什么数据定义成常量？（　　）

A．π　　B．半径　　C．周长　　D．2 和 π

## 2.3　算术运算

描述计算内容和计算过程的公式称为**表达式**（expression）。表达式由**运算符**（operator）、**操作数**（operand）和**括号**（parentheses）组成。C++语言中，在表达式后加分号“;”就构成一条**表达式语句**。表达式语句用于处理数据，是 C++程序中最常用的语句。

运算符有优先级，优先级高的先算。同级运算符按其结合性（从左到右或从右到左）所规定的顺序来计算。括号可以提高优先级，括号内的先算，多层括号时先算里层括号。大部分运算符需要两个操作数，称为双目运算符。某些运算符只需要一个操作数，称为单目运算符。C++语言根据功能和用途将运算符划分为**算术运算符**、**位运算符**、**关系运算符**、**逻辑运算符**等不同类型。本节先介绍算术运算符。

### 2.3.1　C++语言中的加减乘除

加减乘除是最常用的算术运算，C++语言分别用不同的符号来表示它们：+（加）、–（减）、*（乘）、/（除），这些符号称为算术运算符。由算术运算符构成的表达式称为算术表达式。C++语言中加、减、乘、除运算的含义与我们的常识是一致的，但也存在一些区别。

### 1．优先级

运算符有不同的优先级，优先级高的先算。C++语言以 1~15 的数值来表示优先级的高低，1 为最高优先级，15 为最低优先级。例如，*、/的优先级为 3 级，+、–的优先级为 4 级，也就是先乘除，后加减。

### 2．结合性

同级运算符按其结合性（从左到右或从右到左）所规定的顺序来计算。在 C++语言中，不同运算符有不同的结合性。+、–、*、/的结合性是从左到右。也有某些运算符的结合性是从右到左。

### 3．操作数及其数据类型转换

算术表达式中，参与运算的操作数可以是常量、变量等。C++语言中，一个操作数除了代表一个数值，还具有特定的数据类型。例如，表达式“5 + 3”是含两个操作数（常量）的加法运算。这两个操作数的数值分别是 5 和 3，数据类型都是 int 型。在 C++语言中，当不同类型的两个操作数参与算术运算时，要先转换成相同类型，然后再进行计算。C++语言为数据类型转换提供了强制转换和自动转换两种方法。

#### 1）强制转换

数据类型**强制转换**就是由程序员在表达式中将操作数由一种数据类型转换成另一种数据类型。

**C++语法：数据类型强制转换**

| **(数据类型)(操作数) 或 (数据类型)操作数** | |
|---|---|
| 举例： | |
| (short)32 指定 32 为有符号短整型（2 字节） | (long) –32 指定–32 为有符号长整型（4 字节） |
| (float)1.8 指定 1.8 为单精度浮点型（4 字节） | (double)1.8 指定 1.8 为双精度浮点型（8 字节） |
| **注**：数据类型应与操作数的数值相符，否则将造成数值的改变。例如， | |
| (float)32 将 32 变为 32.0（可以接受） | (int)1.8 将 1.8 变为 1(丢失小数部分) |
| (short)32 769 将 32 769 变为–32 767（溢出） | (unsigned short) –1 将–1 变为 65 535(丢失负号） |

说明：为什么(unsigned short)–1 会变成 65 535 呢？因为–1 是负数，其 2 字节补码是 $(11111111\ 11111111)_2$，即 16 位全都是 1。如果让计算机以**有符号格式**来解释这个数，则它是–1；而以**无符号格式**来解释这个数，则它是 65 535。

程序员在 C++程序中编写算术表达式时应合理运用数据类型强制转换。例如表达式“5.5 + 3”，其中 5.5 是 double 型（C++语言默认带小数点的数都是 double 型），3 是 int 型。可以将 3 转换为(double)3，使两个操作数都为 double 型；也可以将 5.5 转换成(int)5.5，使两个操作数都为 int 型。显然后一种转换方法将丢失 5.5 的小数部分，即“(int)5.5 + 3”转换后等价于“5 + 3”，这是不可接受的。通常程序员应采取前一种方法，即“5.5 + (double)3”，其转换后等价于“5.5 + 3.0”。

2）自动转换

程序员在源程序中编写算术表达式时可以将数据类型转换的工作交由编译器程序完成。C++编译器在将 C++源程序编译成目标程序时，如果发现某个算术表达式含有不同类型的操作数，则进行自动转换。**自动转换**（或称为隐含转换）的原则是“将低类型向高类型转换”。C++语言中数据类型的高低顺序如下：

| char | unsigned char | short | unsigned short | int | Unsigned int | long | unsigned long | float | double |
|---|---|---|---|---|---|---|---|---|---|
| 低 | | | | | | | | | 高 |

数据类型越高，其可存储的数值范围越大（因为占用字节数多），精度也越高，因此这种自动转换是安全的。例如表达式“5.5 + 3”，编译器编译时会自动将 3（int 型，低类型）转换为 double 型（高类型），使两个操作数的类型一致，即都为 double 型。“5.5 + 3”经过自动转换，它等价于“5.5 + 3.0”。C++语言的数据类型自动转换功能可以减轻程序员的工作量。

4．表达式结果

在 C++语言中，任何数据都是有数据类型的，因此表达式的计算结果有值，也有数据类型。算术表达式计算结果的数据类型等于其操作数的数据类型。例如，算术表达式

```
5 + 3
```

该表达式计算结果的数值等于 8，而数据类型为 int 型。因为操作数 5 和 3 的数据类型是 int 型，因此表达式结果的数据类型也为 int 型。如果参与运算的操作数类型不同，则进行自动转换。例如，算术表达式

```
5.5 + 3
```

该表达式计算结果的数值等于 8.5，数据类型为 double 型。操作数 5.5 是 double 型，3 是 int 型。遵循“低类型向高类型转换”的原则，3 被自动转换为 double 型。转换后，两个操作数的类型都是 double 型，因此表达式结果的数据类型也为 double 型。两个整数类型相除将丢失小数。例如，算术表达式

```
5 / 2
```

该表达式计算结果的数值不是 2.5，而是 2。因为参与运算的两个操作数都是 int 型，所以表达式结果的数据类型也是 int 型，其数值将丢掉小数而只保留整数部分。将操作数改为 double 或 float 类型可以避免上述丢失小数的问题。例如程序员可以将表达式修改成如下形式：

```
5.0 / 2、5 / 2.0、(double)5 / 2、 5 / (float)2
```

#### 5．括号

在表达式中，括号可以提高优先级。括号内的先算，多层括号时先算里层括号。例如表达式

(3*(2+5) – 1) / 2

与数学上不同的是，C++表达式只使用小括号“( )”，有多层括号时也是这样。C++语言对中括号“[ ]”和大括号“{ }”分别赋予了新的含义，被用在了其他场合。

### 2.3.2 其他算术运算符

C++语言还有几个比较特殊的算术运算符。

#### 1．取正/取负运算符+和–

C++语言中的取正/取负运算符就是数学上所说的添加正负号，可对运算符后面的操作数取正或取负。取正/取负运算符是单目运算符，即只有一个操作数。可以将+32，–32，–x等理解成是一个由取正/取负运算符构成的算术表达式。例如“–x”是一个算术表达式，该表达式结果的数据类型与变量 x 类型相同，数值等于变量 x 中所保存数值的负值。

#### 2．取余运算符%

取余运算是计算两个操作数相除后得到的余数。例如，10÷6 的整数商等于 1，余数等于 4，因此 10 % 6 的结果等于 4。%只能对两个整型操作数进行取余运算，运算结果也是整型。%属于算术运算符，其优先级为 3，结合性为从左到右（与乘除运算符相同）。

#### 3． 自增运算符++

如果想把某个数值型变量 x 的值加 1，可使用自增运算符“++”。例如“x++”，计算机计算该表达式时，先读出变量 x 的值，将其加 1 后重新写回 x 的内存单元。“++”是单目运算符，操作数必须是变量。

“x++”还是一个由自增运算符“++”构成的表达式。该表达式的结果等于 x 加 1 之前的值，数据类型与 x 的类型相同。“++”是一种**泛化**的运算符。与普通运算符相同的是，泛化运算符与操作数一起构成表达式，表达式的结果可以作为操作数继续参与下一步运算。与普通运算符不同的是，泛化运算符在运算的同时还会修改参与运算操作数的值。例如，加减乘除从来不会修改操作数的值，而“x++”在计算表达式结果的同时还会修改操作数 x 的值。C++中类似的运算符还有后续章节将逐步介绍的“– –”“=”等运算符。

将“++”放在变量之后，称为**后置**形式的自增运算符。也可以将“++”放在变量之前，称为**前置**形式的自增运算符。自增运算符的前置与后置存在以下区别：

（1）所构成表达式的结果不同。后置表达式和前置表达式都能将变量的值加 1，但后置表达式的结果等于该变量加 1 之前的值，而前置表达式的结果等于变量加 1 之后的值。例如，已有变量 x：“int x = 10;”则“x++”和“++x”都能将 x 的值加 1，变成 11。但表达式“x++”的结果为 10，而表达式“++x”的结果为 11。表达式结果对该表达式继续参

与下一步计算是有意义的，例如表达式“(x++)*2”的结果等于 20，而表达式“(++x)*2”的结果等于 22。

（2）优先级与结合性不同。后置自增运算符的优先级为 1 级（最高级），结合性为从左到右。而前置自增运算符的优先级为 2 级，结合性为从右到左。

#### 4．自减运算符– –

自减运算符与自增运算符类似，只是将加 1 操作变成减 1 操作。自减运算符也有后置与前置两种形式，例如“x– –”（后置）或“– –x”（前置），其优先级和结合性分别与对应的后置或前置自增运算符相同。

### 本节习题

1．C++语言表达式：5 + 2.0，该表达式结果的数据类型和值分别是（　　）。

A．short，7　　B．int，7　　C．float，7.0　　D．double，7.0

2．C++语言表达式：5 / 2，该表达式结果的数据类型和值分别是（　　）。

A．short，2　　B．int，2　　C．float，2.5　　D．double，2.5

3．C++语言表达式：9 % 5，该表达式结果的数据类型和值分别是（　　）。

A．short，1　　B．int，4　　C．float，1.8　　D．double，4.0

4. 执行 C++语句“int x = 5, y;　y = x++; ”，执行后变量 x 和 y 的值分别为（　　）。

A．5, 5　　B．5, 6　　C．6, 5　　D．6, 6

5. 执行 C++语句“int x = 5, y;　y = – –x; ”，执行后变量 x 和 y 的值分别为（　　）。

A．4, 4　　B．4, 5　　C．5, 4　　D．5, 5

6. 执行 C++语句“int x = 5, y = 6, z;　z = x++ / – –y; ”，执行后变量 x、y 和 z 的值分别为（　　）。

A．5, 5, 1　　B．6, 5, 1　　C．6, 5, 1.2　　D．5, 6, 0

## 2.4　位运算

计算机程序可以用一个**二进制位**来记录某种对象的开关状态，这种二进制位被称为状态位。举个例子，假设用计算机来控制一组电灯（用 1 表示开，0 表示关），则一个字节可以表示 8 盏电灯的开光状态，两个字节就可以表示 16 盏电灯的开光状态。对状态位的设定就可以控制某盏电灯的开关。C++语言提供 6 种位运算符，可应用于状态位的设定或检测，它们被统称为位运算。

#### 1．位反运算符~

“位反”运算符是单目运算符，其运算规则是：将 1 变成 0，0 变成 1。

根据数据类型的不同，程序中参与位反运算的操作数至少有 8 位（char 型）。位反运算是将操作数中的所有位同时进行取反。例如一个 8 位的位反运算：

~　0101 0101
=　1010 1010

在实际应用中，位反运算可将操作数中的所有状态位同时进行反置。假设定义一个无符号字符型变量 s 来记录 8 盏电灯的开关状态：

```
unsigned char s = 0x55;        // 0x55 是十六进制数，其对应的二进制为（0101 0101）2
```

对变量 s 进行位反运算可将 8 盏电灯中原来亮着的灯关闭，原来没亮的灯打开。用 C++语言来描述上述位反运算，其形式为：

```
s = ~s;
```

### 2．位与运算符&

“位与”运算符是双目运算符，其运算规则是：参与运算的两个位都为 1，则结果为 1，否则为 0。参与位与运算的两个操作数是按位进行运算。例如一个 8 位的位与运算：

　　0011 0011
&　0000 1111
=　0000 0011

位与运算可应用于检测操作数中某个状态位的状态，或将其置为 0，此时另一个操作数被称为**掩码**（mask）。假设一个无符号字符型（unsigned char）变量 s，如想检测 s 中某一位的状态是 0 还是 1，则可使用与该位对应的掩码进行位与运算。例如，

　　bbbb bbbb　操作数 s，其中 b 表示 0 或 1
&　0000 0010　检测倒数第 2 位状态的掩码(0x2)
=　0000 00b0　运算结果：保留倒数第 2 位，其他位变成 0
　　　　　　　如果结果为 0(即 8 位全部为 0)，则倒数第 2 位的状态为 0；
　　　　　　　否则倒数第 2 位的状态为 1

用 C++语言来描述上述位与运算，它是一个位与运算表达式，其形式为：

```
s & 0x2
```

位与运算可以将变量 s 中某一位的状态置 0，例如，

　　bbbb bbbb　操作数 s，其中 b 表示 0 或 1
&　1111 1101　将倒数第 2 位状态置 0 的掩码(0xFD)
=　bbbb bb0b　运算结果：将倒数第 2 位置成 0，其他位不变

用 C++语言来描述将变量 s 倒数第 2 位置 0 的操作，它是一条含位与运算的表达式语句，其形式为：

```
s = s & 0xFD;
```

### 3．位或运算符|

“位或”运算符是双目运算符，其运算规则是：参与运算的两个位只要有一位为 1，则结果为 1，否则为 0。参与位或运算的两个操作数也是按位进行运算。例如一个 8 位的位或运算：

　　0011 0011
|　0000 1111
=　0011 1111

位或运算可用于将操作数中的某个状态位置为 1。例如，假设一个无符号字符型（unsigned char）变量 s，则可选择掩码 0x2 将其倒数第 2 位的状态置为 1。

　　bbbb bbbb　操作数 s，其中 b 表示 0 或 1
|　0000 0010　将倒数第 2 位状态置 1 的掩码(0x2)
=　bbbb bb1b　运算结果：将倒数第 2 位置成 1，其他位不变

用 C++语言来描述上述位或运算，其形式为：

```
s = s | 0x2;
```

### 4．异或运算符^

“异或”运算符是双目运算符，其运算规则是：参与运算的两个位不同（0 和 1，或 1 和 0），则结果为 1，否则为 0。参与异或运算的两个操作数按位进行运算。例如一个 8 位的异或运算：

　　0011 0011
^　0000 1111
=　0011 1100

异或运算可用于将操作数中的某个状态位进行反置，即原来为 0 则反置成 1，原来为 1 则反置成 0。例如，假设一个无符号字符型（unsigned char）变量 s，则可选择掩码 0x2 将其倒数第 2 位的状态进行反置。

　　bbbb bb0b　操作数 s，其中 b 表示 0 或 1。假设倒数第 2 位为 0
^　0000 0010　将倒数第 2 位状态进行反置的掩码(0x2)
=　bbbb bb1b　运算结果：将倒数第 2 位由 0 反置成 1，其他位不变

　　bbbb bb1b　操作数 s，其中 b 表示 0 或 1。假设倒数第 2 位为 1
^　0000 0010　将倒数第 2 位状态进行反置的掩码(0x2)
=　bbbb bb0b　运算结果：将倒数第 2 位由 1 反置成 0，其他位不变

用 C++语言来描述上述异或运算，其形式为：

```
s = s ^ 0x2;
```

### 5．左移运算符<<

“左移”运算将操作数按二进制位左移指定的位数，左移时高位被移除，低位补 0。例如将一个 8 位操作数左移 2 位：

　　0011 0011　8 位操作数
<<　2　左移 2 位
=　~~00~~ 1100 1100　高 2 位被移除，低 2 位补 0，得到 1100 1100

左移运算符的语法形式是：

```
操作数 << 左移位数
```

假设一个整型变量 s，用 C++语言来描述将 s 左移 2 位的语法形式是：

```
s << 2
```

### 6．右移运算符>>

“右移”运算将操作数按二进制位右移指定的位数，右移时低位被移除，无符号数高位补 0，有符号数高位补符号位。例如将一个 8 位无符号数右移 2 位：

```
     0011 0011      8 位无符号数
>>   2              右移 2 位
=    0000 1100 11   低 2 位被移除，高 2 位补 0，得到 0000 1100
```

再比如，将一个 8 位有符号数右移 2 位：

```
     1 011 0011     8 位有符号数，最高位为符号位(1 表示负数)
>>   2              右移 2 位
=    1 110 1100 11  低 2 位被移除，高 2 位补符号位 1，得到 1110 1100
```

右移运算符的语法形式是：

```
操作数 >> 右移位数
```

假设一个整型变量 s，用 C++语言来描述将 s 右移 2 位的语法形式是：

```
s >> 2
```

表 2-4 列出了 C++语言 6 种位运算符的优先级和结合性。对比加减运算（4 级）、乘除运算（3 级），除了位反运算，其他位运算符的优先级都比加减乘除要低。

**表 2-4　位运算符的优先级和结合性**

| 位运算符 | 优先级 | 结合性 |
|---|---|---|
| ~，位反 | 2 | 从右向左 |
| <<，左移 | 5 | 从左向右 |
| >>，右移 | | |
| &，位与 | 8 | |
| ^，异或 | 9 | |
| \|，位或 | 10 | |

需要注意的是，所有参与位运算的操作数只能是整型（char、short、int 和 long），包括有符号和无符号格式。如果对其他类型（例如 double）的操作数进行位运算，编译时会提示语法错误。

本节最后再给出一个利用状态位存储数据的例子。假设编写一个闹钟程序，设定工作日（周一~周五）启动闹钟。如何保存哪天启用闹钟的信息呢？可以定义一个如下的变量 AlarmDay：

```
unsigned char AlarmDay;              // 定义一个无符号字符型变量 AlarmDay
```

变量 AlarmDay 占一个字节（8 位），可以用低 7 位分别表示一周中哪几天启用闹钟。1 表示启用，0 表示不启用。最高位未用到，置 0。例如，

```
AlarmDay = 0x1F;                     // 即二进制的 0001 1111，表示周一~周五启用闹钟
```

进一步，可以再定义一组掩码常量来提高程序的可读性。

```
#define MONDAY        0x01     // 即二进制的 0000 0001。最低位为 1，周一启用闹钟的掩码
#define TUESDAY       0x02     // 即二进制的 0000 0010，周二启用闹钟的掩码
#define WEDNESDAY     0x04     // 即二进制的 0000 0100，周三启用闹钟的掩码
#define THURSDAY      0x08     // 即二进制的 0000 1000，周四启用闹钟的掩码
#define FRIDAY        0x10     // 即二进制的 0001 0000，周五启用闹钟的掩码
#define SATURDAY      0x20     // 即二进制的 0010 0000，周六启用闹钟的掩码
#define SUNDAY        0x40     // 即二进制的 0100 0000，周日启用闹钟的掩码
```

则闹钟程序可以用“位或”运算来设置周一~周五启用闹钟。

```
AlarmDay = MONDAY | TUESDAY | WEDNESDAY | THURSDAY | FRIDAY;   // 易于阅读理解
```

## 本节习题

1. 位反运算表达式：~1001，该表达式的结果是（　　）。
   A．–1001　　B．0110
   C．0000　　D．1111
2. 位与运算表达式：1001 & 0110，该表达式的结果是（　　）。
   A．1001　　B．0110
   C．0000　　D．1111
3. 位或运算表达式：1001 | 0110，该表达式的结果是（　　）。
   A．1001　　B．0110
   C．0000　　D．1111
4. 异或运算表达式：1001 ^ 0110，该表达式的结果是（　　）。
   A．1001　　B．0110
   C．0000　　D．1111
5. 执行 C++语句“unsigned char x = 5;　x = x & 0xff;”后变量 x 的值(二进制)为(　　)。
   A．00000101　　B．11111111
   C．00000000　　D．11111010

# 2.5 赋值运算

### 1．赋值运算符

**赋值**（assignment）运算符“=”用于修改变量的数值，即将新数值写入变量对应的内存单元，存储在该内存单元中的原数值将被擦除。例如：

```
int x = 0, y = 0;
x = 5;
y = x + 3;
```

赋值运算符的作用是将“=”右边表达式的结果赋值给左边的变量。常量或变量可以理解成一个最简单的表达式。上例中变量 x，y 的初始值为 0。赋值后，x 的值变成 5，y 的值变成 8。在语法上赋值运算符“=”的左边必须是变量。

赋值运算本身也构成一个赋值表达式。该表达式结果的数据类型与左边变量的类型相同，数值等于左边变量赋值以后的数值。上例中，“x = 5”构成一个赋值表达式，其结果的类型为 int 型（即 x 的数据类型），数值为 5（即 x 赋值以后的数值）。赋值表达式可以继续参与运算，例如，“(x = 5) * 2”的结果等于 10。

赋值运算符的优先级很低（14 级，加减运算符为 4 级），结合性为从右到左。例如，混合运算“y = x = 2 + 6”与“y = ( x = (2+6) )”等价。因为加法优先级高，先算 2+6 得到 8；两个赋值运算符按从右到左的次序先算 x = 8（结果为 8）；最后再算 y = 8（结果也为 8）。计算机执行语句“y = x = 2 + 6;”后，变量 x 和 y 都被赋值为 8。

C++语言中，加、减、乘、除这样的普通运算符在运算时不会改变操作数的值。而赋值运算符“=”和自增自减运算符“++”“– –”一样，属于泛化的运算符，由它们构成的表达式在产生运算结果的同时还会改变操作数的值。合理运用泛化运算符可以让语句更加简洁，例如：

语句：a = 10;　b = 10;　c = 10;　　可写成：a = b = c = 10;
语句：y = x;　x = x + 1;　　可写成：y = x++;
语句：x = x + 1;　y = x;　　可写成：y = ++x;
语句：y = x;　x = x – 1;　　可写成：y = x– –;
语句：x = x – 1;　y = x;　　可写成：y = – –x;

### 2．复合赋值运算符

赋值运算符“=”还可以与部分算术运算符和位运算符组成复合赋值运算符（表 2-5，共 10 种）。

表 2-5　复合赋值运算符

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| += | –= | *= | /= | %= | &= | \|= | ^= | <<= | >>= |

复合赋值运算符的定义是：“x ?= exp”等价于“x = x ? ( exp )”，其中“?”表示某个运算符，x 是一个变量，exp 是一个表达式。“x ?= exp”实际上是“x = x ? ( exp )”的简写形式。例如，

x += 5;　　　等价于　x = x + 5;

计算机执行该语句的过程是：先读出变量 x 的值，与 5 进行加法运算，然后再将运算结果写回 x 对应的内存单元。另外，复合赋值运算符总是先计算右边的表达式，例如，

y *= x + 2;　　等价于　y = y * (x + 2);
y &= x + 2;　　等价于　y = y & (x + 2);
y <<= x + 2;　等价于　y = y << (x + 2);

复合赋值运算符的优先级和结合性与赋值运算符“=”完全一致，即优先级为 14 级，结合性为从右到左。

### 3．变量初始化

定义变量时为变量赋一个初始值，这称为对变量的**初始化**。C++继承了 C 语言使用赋值运算符对变量进行初始化的方法，例如，

```
int x=10, y;
```

该语句定义了两个 int 型变量 x 和 y。计算机执行该语句，将为变量分配内存空间。x 被初始化了，计算机在为 x 分配某个内存单元的同时向该内存单元写入初始值 10。y 没有初始化，通常其初始值是以前程序遗留下来的，是不确定的。

在面向对象程序设计中，变量（被称为对象）初始化是通过构造函数来实现的。C++语言支持面向对象程序设计，可使用面向对象的语法形式来初始化变量。例如上述变量 x 的初始化可改写成如下面向对象的形式：

```
int x(10), y;
```

这条语句同样是将变量 x 的初始值设定为 10。

### 4．常变量

初始化后数值不能改变的变量称为**常变量**（constant variable）。定义时，使用关键字 const 来指定所定义的变量是常变量。

**C++语法：定义常变量**

```
const 数据类型　常变量名 = 初始值 ;
```

语法说明：

- 使用 const 关键字指定常变量。
- 定义常变量时必须初始化。
- 常变量的值不能改变，例如不能被再次赋值。

举例：对比普通变量 y 和常变量 x 的不同

```
int y;                    // 定义普通变量 y
const int x = 5;          // 定义常变量 x，初始值设定为 5
y = x + 5;  cout << x;    // 正确的语法：读取常变量 x。普通变量 y 定义后可以赋值修改
x = 10;  cin >> x;        // 错误的语法：不能改变常变量 x 的值
x = 5;  // 错误的语法：常变量的值不能再次赋值，即使是赋同样的值
```

如果程序所处理的某个数据是常量，在程序运行过程中不需要变动，则可以定义一个常变量来保存该数据。常变量从本质上讲是一个变量，从功能上看就是用变量实现了常量的功能。若对常变量赋值，编译器会提示语法错误。常变量除了具有符号常量的提高程序可读性、便于修改等优点之外，常变量的应用范围更加广泛，这一点将在今后的章节中陆续提到。

### 本节习题

1. 执行语句“int x = 5, y;　y = x / 2;”后变量 y 的数据类型和值分别为（　　）。
   A．int, 2.5　　B．int, 2　　C．double, 2.0　　D．float, 2.5
2. 执行语句“int x = 5;　double y;　y = x / 2;”后变量 y 的数据类型和值分别为(　　)。
   A．int, 2.5　　B．int, 2　　C．double, 2.0　　D．double, 2.5
3. 执行语句“int x = 5;　double y;　y = x / 2.0;”后变量 y 的数据类型和值分别为(　　)。
   A．int, 2.5　　B．int, 2　　C．double, 2.0　　D．double, 2.5
4. 执行语句“int x = 5;　double y = 10.5;　y –= x / 2.0;”后变量 y 的值为（　　）。
   A．2.25　　B．5.0　　C．8.0　　D．8.5
5. 执行语句“int x = 5;　double y = 10.5;　y /= x / 2.5;”后变量 y 的值为（　　）。
   A．2.5　　B．5.0　　C．5.25　　D．12.5

## 2.6 数据的输入与输出

程序的功能是对数据进行处理。通常，原始数据需要用户通过输入设备输入到计算机，处理结果则通过输出设备反馈给用户。以前，操作员在控制台上操作计算机，所运行的程序是命令行程序（不是今天常见的图形界面程序）。控制台主要包括键盘和显示器。操作员通过键盘向计算机下达指令、输入数据，通过显示器查看处理结果。因此人们将键盘称为标准输入，将显示器称为标准输出，将命令行界面程序称为**控制台**（console）程序。这些称呼一直沿用至今。

C++语言将数据从键盘输入到某个内存变量，或将某个内存变量中的数据输出到显示器的过程看作是一种数据流动的过程。站在内存变量的角度，键盘是一种提供输入数据的数据源，显示器则是一种输出数据时的目的地。C++语言将提供输入数据的数据源称作**输入数据流**（input data stream），将输出数据时的目的地称作**输出数据流**（output data stream）。输入数据流和输出数据流统称为**输入/输出流**（I/O stream）。通常，也常将输入/输出简称为I/O。

键盘就是一种输入数据流，C++语言用 cin 表示键盘。显示器则是一种输出数据流，用 cout 表示。输入/输出流不属于 C++语言的主体，是其附属组成部分。使用 cin 和 cout 需要导入一些**外部程序**，导入方法是在程序头部增加如下 2 条语句：

```
#include <iostream>
using namespace std;
```

关于外部程序及其导入方法，将在后续章节进行深入讲解。

**C++语法：标准输入语句**

```
cin >> 变量 1 >> 变量 2 >> …… >> 变量 n ;
```

语法说明：

- cin 表示键盘，借用右移运算符“>>”表示数据从键盘流向后面的变量。
- 一条输入语句可以输入多个变量的数据，输入时用空格或 Tab 键隔开，以回车键结束。
- 键盘所输入数据的类型应与变量的类型匹配。
- 执行该语句时，计算机将暂停程序的执行，等待用户从键盘输入指定个数和类型的数据，然后将这些数据按位置次序赋值给对应的变量。

举例：

```
int x;   double y;
cin >> x;        // 从键盘输入整型变量 x 的值
cin >> x >> y;   // 从键盘输入整型变量 x 和浮点型变量 y 的值
                 // 用户应按次序输入 2 个数据（中间用空格隔开），第 1 个应当是整数，第 2 个应当是实数
```

**C++语法：标准输出语句**

```
cout << 表达式 1 << 表达式 2 << …… << 表达式 n ;
```

语法说明：

- cout 表示显示器，借用左移运算符“<<”表示数据从内存（表达式结果是存放在内存里的）流向显示器。
- 单个常量或变量可认为是最简单的表达式。
- 表达式“endl”表示换行显示。
- 一条输出语句可以同时输出多个表达式结果。
- 执行该语句时，计算机首先按从右到左的顺序逐个计算表达式的结果，然后再按从左到右的顺序依次显示各表达式的结果，各显示结果之间没有间隔。

举例：

```
int x = 5;
cout << x;           // 显示变量 x 的值，显示结果：5
cout << 5;           // 显示一个常量的值，显示结果：5
cout << x << x *x;   // 显示变量 x 及其平方的值（中间没有间隔），显示结果：525
```

例 2-2 给出一个 C++程序例子，其功能是将以克为单位的重量换算成克拉和盎司。在 C++集成开发环境中对该程序进行编译、连接，生成可执行程序。运行这个可执行程序，输入 5，显示器将显示 5 克换算成克拉和盎司的结果（如图 2-1 所示）。

**例 2-2　输入/输出举例：克、克拉与盎司**

```
//   C++程序实例：将以克为单位的重量换算成克拉和盎司
#include <iostream>
using namespace std;

int main( )
{
    double x;                          //  定义一个变量 x 来存放以克为单位的重量
    cin >> x;                          //  从键盘输入需要换算的克数

    cout << x * 5;                     // 1 克  = 5 克拉
    cout << x / 31.1034807;            // 1 盎司  = 31.103 480 7 克
    return 0;
}
```




图 2-1　例 2-2 的运行结果

例 2-2 存在以下两个问题：

（1）输入数据前屏幕没有任何提示信息，程序界面不友好。

（2）两个换算结果克拉和盎司连在一起，中间没有分隔符，很难阅读。

为此我们需要在程序中增加一些提示信息，优化程序的操作界面。C++语言使用**字符串**（详见 4.4 节）来表示这样的提示信息。以双引号括起来的文字序列称为字符串常量，例如：“China”、“中国”等。例 2-3 通过增加一些提示信息来优化程序的操作界面，显示结果见图 2-2。

**例 2-3　界面优化举例：克、克拉与盎司**

```
//   C++程序实例：将以克为单位的重量换算成克拉和盎司
#include <iostream>
using namespace std;

int main( )
{
    double x;                                    //  定义一个变量 x 来存放以克为单位的重量
    cout << "请输入需要换算的重量（以克为单位）：";     //  提示用户正确地输入数据
    cin >> x;                                    //  从键盘输入需要换算的克数

    cout << "换算结果：= " << x * 5 << "克拉，";   // 1 克  = 5 克拉，增加提示信息和分隔符号
    cout << " = " << x / 31.1034807 << "盎司" << endl;     // 1 盎司  = 31.1034807 克
                                                 // endl 表示在显示结束后换一行
    return 0;
}
```

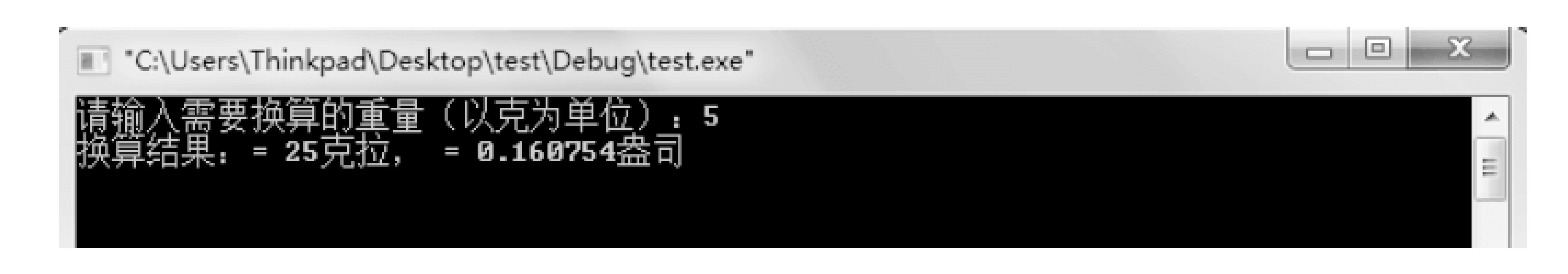



图 2-2　例 2-3 的运行结果

### 本节习题

1. 接收用户从键盘输入的数据并存放到变量 m 中，下列哪条语句是正确的？（　　）
   A．cin >> m;　　B．cin << m;　　C．Cin << m;　　D．cin >> M;
2. 执行语句“double x;　cin >> x;”，下列哪种键盘输入是错误的？（　　）
   A．5　　B．5.0　　C．x=5.0　　D．0.5e1
3. 执行语句“int x;　double y;　cin >> x >> y;”，下列哪种键盘输入是正确的？(　　)
   A．5, 10.5　　B．5 10.5　　C．5.0, 10.5　　D．5.0 10.5
4. 执行语句“int x = 5, y = 10;　cout << x << y;”，则显示器将显示（　　）。
   A．5 10　　B．5, 10　　C．10, 5　　D．510
5. 执行语句“int x = 5, y = 10;　cout << x << ", " << y;”，则显示器将显示（　　）。
   A．5 10　　B．5, 10　　C．10, 5　　D．510
6. 执行语句“int x = 5;　cout << x << "，" << x++;”，则显示器将显示（　　）。
   A．5, 5　　B．5, 6　　C．6, 5　　D．6, 6

## 2.7　引用与指针

计算机程序利用内存来存放数据。数据要存放在内存中才能被 CPU 读取和处理，处理后的结果也只能保存回内存中。程序员通过 C++语言的定义变量语句来申请所需的内存空间。例如，一个程序可能需要定义若干个变量来分别保存原始数据、中间结果和最终结果。

程序员定义变量时须指定变量名，然后通过变量名访问其对应的内存单元（例如写入数据或读出数据）。简单地说，程序中的**变量=内存单元**。变量名是访问变量内存单元的第一种方法。本节将再介绍另外两种方法，它们分别是引用和指针。

### 2.7.1　引用

C++语言允许为已定义的变量再起一个别名，称为变量的引用名。引用名看起来像是一个变量名，但它是一种特殊变量，称为引用变量，或简称为**引用**（reference）。引用变量与其所引用的变量共用同一个内存单元，定义引用变量时不再单独分配内存空间。

### C++语法：定义引用变量

**引用类型 &引用变量名 = 被引用变量名 ；**

语法说明：

- **引用类型**是引用变量的数据类型，必须与被引用变量的类型一致。
- &是引用变量说明符。定义变量时，变量名前加“&”表示该变量为引用变量。
- **引用变量名**需符合标识符的命名规则。
- **被引用变量名**指定一个已经定义的变量，即被引用的变量。
- 定义引用变量时必须初始化，即指定其是哪个变量的引用。引用变量只能引用一个变量，定义后不能再引用其他变量。

举例：定义一个 int 型变量 x 及其引用变量 xa

```
int x ;   int &xa = x;              // xa 是 x 的一个引用，即别名
```

或

```
int x, &xa = x ;                    // 可在一条定义语句中完成
int x;   int y, &xa = x;            // 一条定义变量语句可既包括普通变量，又包括引用变量
```

假设有如下定义变量语句：

```
int x=10, y=20;   int &xa = x;
```

计算机执行上述语句后，所分配的内存空间如图 2-3 所示。

| | |
|---|---|
| | |
| 变量名x<br>引用名(别名)xa | 10<br>(4个字节) |
| 变量名y | 20<br>(4个字节) |
| | …… |

图 2-3　变量 x 及其引用变量 xa

在定义变量 x 的引用之后，引用名 xa 和原变量名 x 所表示的是同一个内存单元，访问效果是一样的。

下面例 2-4 中，代码第 14 行使用引用名 xa 替换原变量名 x（被注释掉的第 10 行代码），计算结果是一样的。

**例 2-4　引用变量举例**

```
1  //  C++程序实例：从键盘输入一个数值，计算其平方的值
2  #include <iostream>
3  using namespace std;
```

```

int main( )
{
    int x;                          // 定义一个变量 x
    cin >> x;                       // 从键盘输入 x 的值
    /*
    cout << x * x << endl;          // 计算并显示 x 的平方值
    */

    int &xa = x;                    // 定义一个变量 x 的引用变量 xa
    cout << xa * xa << endl;        // 改用访问变量 xa 来计算并显示 x 的平方值
                                    // 访问引用变量 xa 所读出的数据就是变量 x 的值
    return 0;
}
```

### 2.7.2 指针

计算机对内存进行读写操作的最小单位是字节。为每个字节指定一个整数编号（通常从 0 开始，连续编号），称为该字节的内存地址。程序执行时，计算机将系统中的空闲内存分配给程序中定义的变量。C++语言提供一个**取地址运算符“&”**来获取变量的内存地址。

**C++语法：取地址运算符&**

**&变量名**

语法说明：

- 所取出的变量地址是程序执行时该变量所分配内存单元的地址。每次执行程序时，变量不一定会被分配在同一内存单元，这取决于本次执行时计算机中哪些内存单元是空闲的。
- 一个变量可能占用多个字节。变量地址指的是变量所占内存单元第一个字节的地址，也称首地址。
- 取地址运算符是单目运算符，操作数必须是变量，其优先级为 2 级，结合性为从右向左。
- 在 C++语言中，“&”是一符多义的符号——位运算中的位与运算符、定义变量语句中的引用变量说明符、取地址运算符。不同场合具有不同的含义，程序员应根据上下文来区分。

举例：已定义变量 x：int x = 10;

```
cout << x;          // 显示变量 x 内存单元中保存的数值：10
cout << &x;         // 显示变量 x 的内存地址
```

例如，程序定义了一个变量 x“int x=10;”，程序执行时将为 x 分配 4 个字节。这 4 个字节是连续的，假设其地址为 1000~1003，则变量 x 的地址就是其首地址 1000，参见图 2-4。

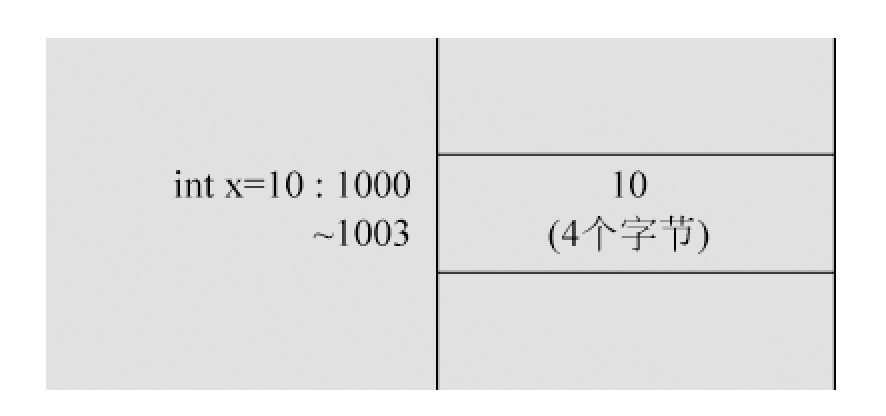



图 2-4　变量的内存地址

执行输出语句：

```
cout << x;
```

将显示变量 x 内存单元中保存的数值：10

而执行输出语句：

```
cout << &x ;
```

将显示变量 x 的内存地址：000003E8（十六进制的 1000）。

内存地址是一类特殊类型的数据，C++语言将地址类型称为指针类型，或简称为**指针**（pointer）。在 32 位计算机系统中，指针类型的存储位数为 32 位（4 字节），并以无符号整数形式存储。C++语言可以通过内存地址来访问内存单元。

给定某个内存地址，该如何访问内存单元呢？内存单元中存放的可能是一个 int 型数据，也可能是一个 double 型数据。不同数据类型所占用的字节数不同，存储格式也不同。通过地址访问内存单元时，需要知道该地址对应了什么样的数据类型，这个数据类型被称为地址的**指向类型**。例如，通过一个指向类型为 int 型的地址去读取某个内存单元时，计算机将按照 int 型的规定读取 4 个字节，并按补码格式来解释所读出的二进制数据；而通过一个指向类型为 double 型的地址去读取某个内存单元时，计算机将按照 double 型的规定读取 8 个字节，并按浮点格式来解释所读出的二进制数据。

通过地址访问某个变量 x 的步骤一般分为 3 步：首先定义一个专门保存地址的变量（假设为 p），该变量称为指针变量；取出变量 x 的地址，将其赋值给 p；通过指针变量 p 来访问变量 x 的内存单元。

### 1．指针变量

首先我们通过一个例子来观察什么是指针变量。先定义变量 x 和 y：

```
short x=10, y=20;
```

变量 x 占用 2 个字节（short 型，假设地址分别为 1000 和 1001）；变量 y 也占用 2 个字节（假设地址分别为 1002 和 1003）。再定义一个指针变量 p，p 占用 4 个字节（假设地址为 2000~2003）。p 是专门保存其他变量地址的指针变量。如果指针变量 p 中保存的是变量 x 的地址 1000，则称指针变量 p **指向**变量 x（如图 2-5 所示）。p 是变量，可保存不同变量

的地址。如将 p 中保存的地址修改为变量 y 的地址（1002），则我们称指针变量 p 改变了指向，指向了 y。只要指针变量保存某个变量的地址，我们就形象地说指针变量指向了该变量，这也是地址类型被称作指针类型的原因。

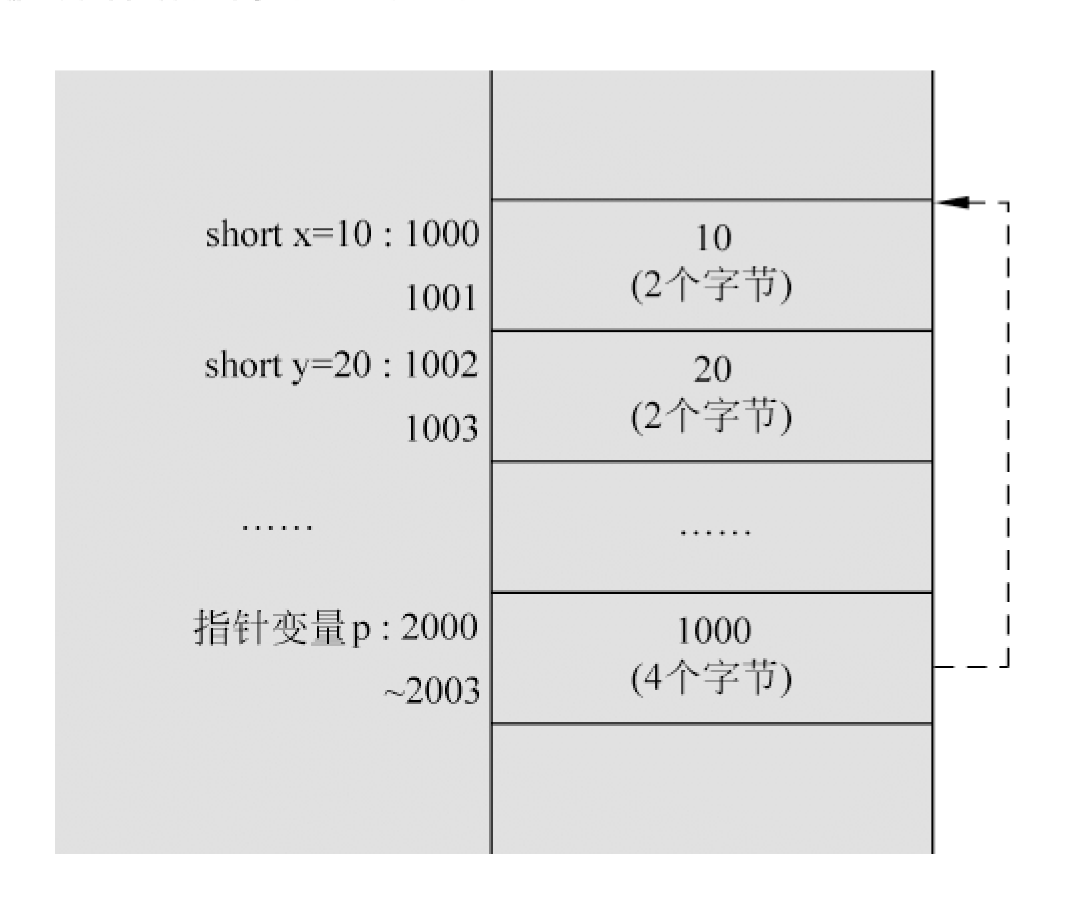



图 2-5　指针变量示意图

一个指针变量虽然能指向不同的变量，但只能指向同一数据类型的不同变量，这个数据类型就是指针变量的指向类型。例如，一个指向类型为 int 型的指针变量（简称为 int 型指针变量），该指针变量只能指向 int 型变量，即只能保存 int 型变量的地址。相应地，一个指向类型为 double 型的指针变量（简称为 double 型指针变量），该指针变量只能指向 double 型变量。

### C++语法：定义指针变量

**指向类型　*指针变量名**;

语法说明：

- **指向类型**指定了指针变量能够保存哪种类型变量的地址，或者说指定了指针变量能够指向哪种类型的变量。
- *是指针变量说明符。定义变量时，变量名前加“*”表示该变量为指针变量。
- **指针变量名**需符合标识符的命名规则。

举例：假设已定义变量 x 和 y：int x, y;

```
int *p;          // “int *”表示 int 型指针
                 // 定义一个 int 型指针变量 p，未初始化（即未指向任何变量）
p = &x;          // 取出变量 x 的地址并赋值给指针变量 p，此时 p 指向了变量 x
```

或

```
int *p = &x;     // 定义一个 int 型指针变量 p，初始化为指向变量 x
p = &y;          // 取出变量 y 的地址并赋值给指针变量 p，则 p 修改了指向，现指向变量 y
```

### 2．变量的间接访问

定义一个变量 x 之后，通过变量名 x 来访问该变量内存单元被称为**直接访问**；为变量 x 定义一个引用变量 xa，通过引用名 xa 来访问变量 x 内存单元是一种间接访问的形式，被称为变量的**间接访问**；定义一个指针变量 p 保存变量 x 的地址，再通过指针变量 p 中所保存的地址来访问变量 x 的内存单元，这是另一种形式的间接访问。为了实现通过内存地址间接访问变量，C++语言提供了一个**指针运算符**（或称为**取内容运算符**）“*”。

**C++语法：指针运算符***

***指针变量名**

语法说明：

- 按照指针变量所保存的地址间接访问所指向的内存单元，可写入或读出数据。访问时将按照指针变量指向类型所规定的字节数和存储格式去读/写内存单元。
- 间接访问之前，指针变量应当指向某个已经存在的变量，即指针变量必须先赋值，再间接访问，否则将出现错误。
- 指针运算符是单目运算符，其优先级为 2 级，结合性为从右向左。
- C++语言中，“*”是一符多义的符号——算术运算中的乘法运算符、定义变量语句中的指针变量说明符、指针运算符。不同场合具有不同的含义，程序员应根据上下文来区分。

举例：假设已定义变量 x：int x;

```
x = 10;          // 通过变量名直接访问，将变量 x 内存单元中的数值修改为 10
cout << x;       // 通过变量名直接访问，显示变量 x 内存单元中保存的数值：10
```

或

```
int *p = &x;     // 定义一个与变量 x 数据类型一致的指针变量 p，初始化为指向变量 x
*p = 10;         // 通过指针变量间接访问 x，将变量 x 内存单元中的数值修改为 10
cout << *p;      // 通过指针变量间接访问 x，显示变量 x 内存单元中保存的数值：10
cout << p;       // 直接访问指针变量自身，显示 p 中所保存的地址（即变量 x 的地址）
```

例 2-5 中的代码第 14 行通过间接访问形式“*p”来替换原来的变量名 x 直接访问（被注释掉的第 10 行代码），计算结果是一样的。

**例 2-5　指针变量举例**

```
1   //  C++程序实例：从键盘输入一个数值，计算其平方值
2   #include <iostream>
3   using namespace std;
4
5   int main( )
6   {
7       int x;                    // 定义一个变量 x
8       cin >> x;                 // 从键盘输入 x 的值
9       /*
10      cout << x * x << endl;    // 计算并显示 x 的平方值
11      */
```

```

    int *p = &x;                    // 定义一个指针变量 p，初始化为指向变量 x
    cout << (*p) * (*p) << endl;    // 改用间接访问*p 来计算并显示 x 的平方值
                                    // 间接访问*p 所读出的数据就是变量 x 的值
    return 0;
}
```

取地址运算符“&”和指针运算符“*”的优先级都为 2 级，结合性为从右向左。例如，表达式：

```
*p * *p
```

与表达式：

```
(*p) * (*p)
```

等价。在上述表达式中，C++语言会根据上下文自动将指针变量之前的“*”当做指针运算符，而将中间的那个“*”当做乘法运算符。指针运算符优先级高（2 级），乘法运算符低（3 级），因此先计算指针运算符*p，然后再计算乘法。

### 3．使用指针变量应当注意的问题

（1）**指针变量应当先赋值，再间接访问**。例如，执行以下代码将出现错误：

```
int *p;
cout << *p;     // 错误：未初始化的指针变量 p 中可能保存了一个随机的地址值
                // 间接访问该地址所对应的内存单元将出现不可预料的错误
```

一个多任务操作系统（例如 Windows）可以同时运行多个程序。每个程序只能访问自己所分配的内存单元。随意访问其他程序的内存单元，或访问一个不存在的内存单元，这都属于严重错误，是被严格禁止的。正确的做法是：先对指针变量赋值，指向一个本程序已定义的变量（该变量已被分配内存空间），然后才可以间接访问该变量。

（2）**指针变量不能用整数来赋值**。例如：

```
int *p = 10;    // 错误：指针变量不能用整数来赋值
```

可以将指针变量赋值为 0。0 表示**空地址**，即不指向任何变量，例如：

```
int *p = 0;     // 正确：赋值为 0，表示 p 不指向任何变量
```

（3）**指针变量的指向类型应当与所指向变量的类型一致**。例如，编译以下代码将出现错误：

```
double x = 10.5;
int *p;
p = &x;         // 将 int 型指针变量 p 指向 double 型变量 x，编译时将提示错误
```

正确的做法是将指针变量 p 定义语句中的指向类型 int 改为 double，与变量 x 的数据类型一致。

C++语言还提供了一种特殊的指向类型，称为 void **类型**。该类型的含义是，所指向变量的数据类型是未知的。void 型指针变量可以指向任意类型的变量。如果使用 void 型指针变量间接访问变量，则访问时需将其指向类型强制转换成所指向变量的类型。例如：

```
int x = 10;   double y = 10.5;
void *p;                // 定义一个 void 型指针变量 p，p 可指向任意类型的变量
p = &x;                 // 将 void 型指针变量 p 指向 int 型变量 x
cout << *( (int *)p );  // 通过 void 型指针变量 p 间接访问 int 型变量 x 时，需要将指针
                        // 变量 p 的指向类型强制转换成 int，即(int *)p。显示结果为 10
p = &y;                 // 修改 void 型指针变量 p 的指向，改为指向 double 型变量 y
cout << *( (double *)p ); // 通过 void 型指针变量 p 间接访问 double 型变量 y 时，
                        // 需将指针变量 p 强制转换成 double 型，即(double *)p。显示结果为 10.5
```

（4）**相同类型指针变量之间可以相互赋值**。可以将一个指针变量的地址值赋给另一个相同类型的指针变量。可以将任意类型指针变量的地址值赋值给一个 void 型指针变量。例如：

```
int x = 10, *pi = &x;       // 定义一个 int 型指针变量 pi，初始化指向变量 x
double y = 10.5, *pd = &y;  // 定义一个 double 型指针变量 pd，初始化指向变量 y

int *p1;                    // 定义一个 int 型指针变量 p1（未初始化），可指向任意 int 型变量
void *p2;                   // 定义一个 void 型指针变量 p2（未初始化），可指向任意类型的变量
p1 = pi;                    // 正确：将 pi 赋值给 p1。两者都为 int 型，赋值后都指向 int 型变量 x
p1 = pd;                    // 错误：不能将 double 型的 pd 赋值给 int 型 p1，类型不同不能互相赋值
p2 = pi;                    // 正确：可将 int 型的 pi 赋值给 void 型 p2，赋值后都指向变量 x
p2 = pd;                    // 正确：可将 double 型的 pd 赋值给 void 型 p2，赋值后都指向变量 y
```

（5）**可以定义指向常变量的指针，通过该指针变量不能修改所指向的变量的值**。初始化后数值不能改变的变量称为常变量。定义指向常变量的指针变量时，需在数据类型之前加 const 关键字。例如：

```
const int x = 10;           // 定义一个 int 型常变量 x，初始值为 10
x = 15;                     // 错误：常变量不能再被赋值

const int *p;               // 定义一个指针变量 p，该指针变量可指向 int 型常变量或普通变量
p = &x;                     // 将 p 指向常变量 x
cout << *p;                 // 正确：通过指针变量 p 间接读取常变量 x 的值
*p = 15;                    // 错误：不能通过指针变量间接修改常变量 x 的值

int y = 20;                 // 定义一个 int 型的普通变量 y，初始值为 20
p = &y;                     // 将 p 指向普通变量 y
cout << *p;                 // 正确：通过指针变量 p 间接读取变量 y 的值
*p = 15;                    // 错误：不能通过 p 间接修改变量 y 的值
                            // 尽管 y 只是一个普通变量，但 p 被定义为指向常变量的指针
```

（6）**可以定义指针类型的常变量（即指针常变量），需定义时初始化，以后不能再改变其指向**。定义指针类型的常变量时，需在变量名之前加 const 关键字。例如：

```
int x = 10, y = 20;          // 定义 int 型变量 x 和 y
int * const p = &x;          // 定义 int 型指针变量 p，该指针变量为常变量，指向变量 x
                             // 常变量在定义时必须初始化，初始化后不能改变指向
p = &y;                      // 错误：常变量不能再次赋值，因此初始化后不能修改常变量 p 的指向
```

爱思考的读者可能会问：既然可以通过变量名直接访问内存单元，为什么 C++语言还要设计这么复杂的语法，通过引用或指针对变量进行间接访问呢？这里我们给出一个最通俗的解释。程序员通过定义变量来申请内存，再用变量名访问所分配的内存单元。大型程序需要多个程序员协作开发，共同完成。如果其他程序员想访问上述变量的内存单元，例如读取其中的数据，可以吗？答案是肯定的，但只能通过引用或指针来访问（即间接访问），关于这一点将在第 5 章做详细讲解。

## 本节习题

1．下列定义引用变量 rx 的语句中，正确的是(　　)。

A．int x;　int rx = x;　　B．int x, &rx = x;　　C．int x, rx = &x;　　D．int &rx = x, x;

2．执行语句“int x = 5, &y = x;　y = x + 10;”后变量 x 的值为(　　)。

A．5　　B．10　　C．15　　D．20

3．下列定义指针变量 px 的语句中，正确的是(　　)。

A．int x;　int px = x;　　B．int x, *px = x;

C．int x, *px = &x;　　D．int *px = &x, x;

4．执行语句“int x = 5, *y = &x;　*y = x + 10;　”后变量 x 的值为(　　)。

A．5　　B．10　　C．15　　D．20

5．下列定义并使用指针变量 px 的语句中，正确的是(　　)。

A．int x, *px;　px = 10;　　B．int x, *px = &x;　px = 10;

C．int x, *px;　*px = 10;　　D．int x, *px = &x;　*px = 10;

6．执行下列 C++程序：

```
int x = 5, *y = &x;
cout << x * (*y);
```

显示器将显示(　　)。

A．5　　B．25　　C．55　　D．不确定

7．下列定义并使用指向常变量 x 的指针变量 px 的语句中，正确的是(　　)。

A．const int x = 10;　const int *px = &x;　*px = 20;

B．const int x = 10;　const int *px;　px = x;

C．const int x = 10;　const int *px = &x;　(*px)++;

D．const int x = 10;　const int *px = &x;　cout << *px;

## 学习本章的要点

■ 读者需将程序中的数据与内存联系起来，这样就很容易理解数据类型、引用和指针等初学者难以掌握的概念。

- 读者重点要关注运算符的运算规则、优先级和结合性等语法细节。
- 本章会让读者初步体会到计算机语言与人类语言的不同之处，即计算机语言的语法规则非常严格，甚至到了机械的程度，稍有不慎就会出现语法错误。

## 2.8 本章习题

**1．阅读程序**。阅读下列 C++程序。阅读后请说明程序的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
using namespace std;
#define PI   3.14
int main( )
{
    float   r;
    cin >> r;
    float len;
    len = PI * 2 * r;
    cout << "len= " << len << endl;
    return 0;
}
```

**2．程序改错**。阅读下列 C++程序，并检查其中的语法错误。修改错误，并保证程序的功能不变。

```
#include <iostream>
using namespace std;
int main( )
{
    int x = y = 5;                         // 定义两个变量 x、y，初始值都为 5
    cout >> x, y >> endl;                  // 显示 x 和 y 的值，并用逗号隔开
    int z, rz = &z;                        // 定义变量 z 及其引用变量 rz
    z = x ÷ y;                             // 求 x 除以 y 的余数，并赋值给 z
    cout << "余数是" << &rz << endl;       // 通过引用变量 rz 显示变量 z 的值

    int pz = z;                            // 定义指向变量 z 的指针变量 pz
    pz = x * y;                            // 求 x 乘以 y 的积，并通过指针变量 pz 存入变量 z 中
    cout << "乘积是" << pz << endl;        // 通过指针变量 pz 显示变量 z 的值
    return 0;
}
```

**3．编写程序**。请编写一个计算表达式 $x\{2+5[3x^2+8(x-1)+6]\}$ 的 C++程序。

# 第3章 算法与控制结构

一个完成某种特定任务的过程可分解成一组操作步骤，这组操作步骤即构成一个算法。算法是一个宽泛的概念，求解数学问题要用到算法，日常生活中也经常用到算法。例如，制作回锅肉的菜谱就可以认为是一个算法，如例 3-1 所示。

| 例 3-1　算法举例：制作回锅肉的菜谱 | |
|---|---|
| **原料** | 主料：400g 五花肉、250g 青蒜<br>配料：适量葱、姜、蒜、干红辣椒，1 勺花椒、1 大勺郫县豆瓣酱，适量料酒、糖、酱油 |
| **做法** | |
| 1 | 带皮五花肉冷水下锅，加入葱段、姜片、花椒 7~8 粒、黄酒适量，煮开。 |
| 2 | 撇净浮沫，煮至八成熟，取出自然冷却。 |
| 3 | 将肉切成薄片，姜、蒜切片，葱切成斜段。 |
| 4 | 将青蒜的白色部分先用刀拍一下，然后全部斜切成段备用。 |
| 5 | 炒锅上火，加少量的油煸香辣椒、花椒及葱、姜、蒜。 |
| 6 | 下入肉片煸炒，至肉片颜色变得透明，边缘略微卷起。 |
| 7 | 将肉拨到锅一边，下入郫县豆瓣酱（可以先剁细）炒出红油。 |
| 8 | 适当加入少许酱油或甜面酱调色，与肉片一起翻炒均匀。 |
| 9 | 下入青蒜，点少许料酒、糖，调好味道即可出锅，见图 3-1。 |



图 3-1　经典川菜——回锅肉

将从原料到回锅肉的制作过程分解成若干个步骤。每个步骤简单具体，具有可操作性，是任何人都可以掌握的。菜谱就是一种算法。按照这个算法进行操作，就可以制作出一道美味可口的回锅肉。

程序设计课程关注的是能被计算机执行的算法。本章讨论程序设计中算法的概念、基本结构以及相关的 C++语句。

## 3.1 算法

程序设计中，程序员将完成某种程序功能的过程分解成一组可被计算机执行的操作步骤，这组操作步骤称为**算法**（algorithm）。例如，为了使用计算机将摄氏温度换算成华氏温度，程序员需要为计算机设计一个温度换算算法，如例 3-2 所示。

| 例 3-2　算法举例：将摄氏温度换算成华氏温度 | |
|---|---|
| 1 | 定义变量，申请保存摄氏温度和华氏温度数据所需的内存空间。 |
| 2 | 从键盘输入需要换算的摄氏温度，将数据保存到摄氏温度变量中。 |
| 3 | 换算公式：华氏温度=摄氏温度×1.8＋32，将换算结果保存到华氏温度变量中。 |
| 4 | 在显示器上显示换算得到的华氏温度。 |

可以用多种方法来描述算法设计的结果。常用的有流程图、伪代码或自然语言。例 3-2 是用自然语言描述的温度换算算法，而图 3-2 是用流程图来描述该算法。

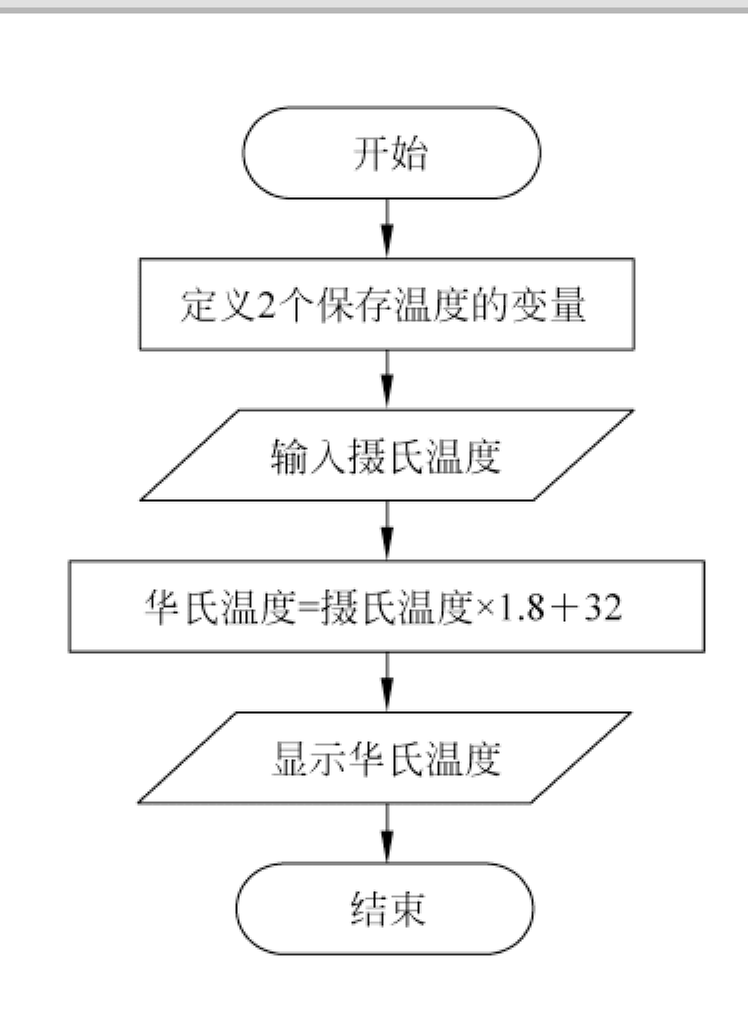


图 3-2　温度换算算法流程图

按书写顺序依次执行操作步骤的算法称为**顺序结构**算法。例 3-2 就是一种顺序结构算法。算法有三种基本结构，分别是顺序结构、选择结构和循环结构。顺序结构是最简单的一种算法结构。算法中，某些操作步骤需要满足特定条件才被执行，这种算法结构称为**选择结构**。还有一些算法，在满足特定条件下将重复执行某些操作步骤，这种算法结构称为**循环结构**。

在上述三种算法结构中，选择结构和循环结构都要用到**条件**。如果一个条件成立，我们称这个条件为**真**，否则称之为**假**。C++语言使用布尔类型来表示条件的真假，通过关系运算符（例如大于、小于、等于）构成的关系表达式来描述一个条件，通过逻辑运算符（与、或、非）构成的逻辑表达式来描述一个复合条件。

使用 C++语言将设计好的算法编写成一组语句序列，这就是 C++源程序。为了描述选择结构和循环结构的算法，C++语言分别提供了选择语句和循环语句。

### 本节习题

1. 将数据处理的过程分解成一组操作步骤，这种操作步骤被称为（　　）。
   A. 数据　B. 算法　C. 程序　D. 流程图
2. 一个计算机程序主要由数据和（　　）两部分内容组成。
   A. 输入　B. 输出　C. 公式　D. 算法
3. 描述算法有几种常用的方法，下列哪种方法不属于常用方法？（　　）
   A. 流程图　B. 中文　C. 图纸　D. 伪代码
4. 下列哪种结构不属于算法的三种基本结构？（　　）
   A. 顺序结构　B. 并列结构　C. 选择结构　D. 循环结构
5. 下列哪种算法结构不需要条件？（　　）
   A. 顺序结构　B. 选择结构
   C. 循环结构　D. 任何算法结构都不需要

## 3.2 布尔类型

选择结构和循环结构都需要用到条件。如果一个条件成立，我们称这个条件为真，否则称之为假。C++语言使用**布尔类型**（bool）来表示条件的真假。布尔类型的取值只有两个，即 true 和 false，其中 true 表示真，false 表示假。bool、true 和 false 都是 C++语言的关键字。

可定义 bool 型变量来保存 bool 型数据。一个 bool 型变量占用 1 个字节。由于计算机只能存储数值数据，因此计算机内部存储 bool 型数据时以 1 来表示 true，0 表示 false。例 3-3 演示了 bool 类型的应用，以及 bool 类型与其他数值类型之间的转换。

**例 3-3　bool 类型应用举例**

```
#include <iostream>
using namespace std;

int main( )
{
    bool x = true;   // 定义一个 bool 型变量 x，并初始化为 true。true 和 false 是 bool 型常量
    cout << x << endl;   // 显示变量 x 的值，true 被显示为 1

    int y;    // 再定义一个 int 型变量 y
    y = x;    // 将 bool 型变量 x 赋值给 int 型变量 y，C++将自动转换类型，true 被转换为 1
    cout << y << endl;   // 显示变量 y 的值，显示结果为 1

    x = 5;   // 将 int 型常量 5 赋值给 bool 型变量 x，5 被转换为 true，即非 0 值转为 true
             // 此时编译系统会提示警告性（warning）错误
    cout << x << endl;     // 显示变量 x 的值，true 被显示为 1
    return 0;
}
```

bool 类型的应用说明如下：

- true 和 false 是两个 bool 型常量，可直接在程序中使用，例如代码第 6 行。
- 计算机内部存储 bool 型数据时以 1 来表示 true，0 表示 false，如代码第 7 行。
- 将 bool 类型转为其他数值类型时，true 转为 1，false 转为 0，如代码第 10 行。
- 将其他数值类型转为 bool 类型时，0 转为 false，非 0 值转为 true，如代码第 13 行。

C++语言通过关系运算符构成的关系表达式来描述一个条件，通过逻辑运算符构成的逻辑表达式来描述一个复合条件。关系表达式和逻辑表达式的结果都是 bool 类型。

### 3.2.1 关系运算符

C++语言提供 6 个关系运算符，用于比较两个数之间的大小，见表 3-1。

**表 3-1 关系运算符**

| 关系运算符 | 优先级 | 结合性 |
|---|---|---|
| > （大于）<br>>=（大于等于）<br>< （小于）<br><=（小于等于） | 6 | 从左向右 |
| ==（等于）<br>!= （不等于） | 7 | |

由关系运算符构成的表达式称为**关系表达式**，其运算结果是布尔类型。例 3-4 列举了一些关系表达式的例子。

**例 3-4 关系表达式举例**

| 关系表达式 | 布尔型结果 | 备注 |
|---|---|---|
| 5 > 3 | true | |
| 5 >= 3 | true | |
| 5 <= 3 | false | |
| 5 == 3 | false | |
| 5 != 3 | true | |
| 2+3 <= 1+2 | false | 比较两个算术表达式时，先计算表达式，再比较其结果。算术运算符优先级高于关系运算符 |

选择结构或循环结构中的条件通常用于判断程序中变量当前数值的大小。假设已定义变量 x：

```
int x = 10;
```

则例 3-5 中的关系表达式都可以构成一个条件。

例 3-5　由关系表达式所描述的条件举例（假设：int x = 10;）

| 条件 | 布尔型结果 | 条件是否成立 |
|---|---|---|
| x > 5 | true | 条件成立 |
| x < 5 | false | 条件不成立 |
| x − 5 == 5 | true | 条件成立 |
| x − 5 < 0 | false | 条件不成立 |

## 3.2.2　逻辑运算符

C++语言提供 3 个逻辑运算符，用于将多个条件组合成一个复合条件，见表 3-2。

表 3-2　逻辑运算符

| 逻辑运算符 | 优先级 | 运算规则 |
|---|---|---|
| &&（逻辑与） | 11 | 双目运算符。若两个操作数都为 true，则结果为 true；否则为 false。相当于“**并且**”的意思 |
| \|\| （逻辑或） | 12 | 双目运算符。若两个操作数中有一个为 true，则结果为 true；否则为 false。相当于“**或**”的意思 |
| ! （逻辑非） | 2 | 单目运算符。若操作数为 true，则结果为 false；若操作数为 false，则结果为 true。相当于“**求反**”的意思 |

由逻辑运算符构成的表达式称为**逻辑表达式**，其运算结果是布尔类型。假设已定义变量 x 和 y：

```
int x = 10, y = 20;
```

则例 3-6 中的逻辑表达式都可以构成一个复合条件。

例 3-6　由逻辑表达式所描述的复合条件举例（假设：int x=10, y=20;）

| 复合条件 | 布尔型结果 | 条件是否成立 |
|---|---|---|
| x > 5 && y > 10 | true | 条件成立 |
| x < 5 \|\| y < 10 | false | 条件不成立 |
| x − 5 == 5 \|\| y == 0 | true | 条件成立 |
| !(x > 5) | false | 条件不成立 |

参与逻辑运算的操作数要求是布尔类型，否则将自动转换成布尔类型。其他类型数据转为布尔类型的规则是：0 转为 false，非 0 值转为 true。例如，表达式“0 && true”的结果为 false，因为 0 被自动转换为 false。而表达式“5 && true”的结果为 true，因为 5 被自动转换为 true。

## 本节习题

1. bool 类型的存储位数与下列哪种数据类型相同？（　　）

A. char　　B. short　　C. long　　D. float

2. 若有 C++语言表达式 5 > 9，则该表达式结果的数据类型和值分别是（　　）。

A. int，5　　B. int，9　　C. bool，true　　D. bool，false

3. 若有 C++语言表达式 5 <= 5，则该表达式结果的数据类型和值分别是（　　）。

A. int，0　　B. int，true

C. bool，true　　D. bool，false

4. 比较变量 x 的值是否等于 5，下列哪个表达式是正确的？（　　）

A. x = 5　　B. x == 5　　C. x <> 5　　D. x ~ 5

5. 若有 C++语言表达式 1 >= 0 && 0 <= 1，则该表达式的结果是（　　）。

A. 0　　B. 1　　C. true　　D. false

6. 下列哪个表达式的结果为 true？（　　）

A. !(5 > 1)　　B. 5 > 1 && false

C. 5 > 1 || false　　D. 5 < 1 || false

7. 若有 C++语言表达式 5 && true，则该表达式结果的数据类型和值分别是（　　）。

A. int，5　　B. int，true

C. bool，true　　D. bool，false

## 3.3 选择语句

在有些算法中，某些操作步骤需要满足特定条件才被执行。例如，给定 x 的值，求其倒数。当 x=0 时，倒数 1/x 没有意义。因此在设计求倒数算法时，应当判断条件“x 不等于 0”是否成立。如果成立则求 x 的倒数，否则应提示错误信息。具体的求倒数算法见例 3-7。

| 例 3-7　算法举例：给定 x 的值，求其倒数 | |
| --- | --- |
| 1 | 定义变量 x，申请保存数值的内存空间。 |
| 2 | 从键盘输入变量 x 的值。 |
| 3 | 如果条件“x 不等于 0”成立，则转到步骤 4 计算倒数，否则转到步骤 5 提示错误信息。 |
| 4 | 计算并显示表达式 1/x 的结果，转到步骤 6。 |
| 5 | 条件“x 不等于 0”不成立（即 x 等于 0），显示错误信息。 |
| 6 | 算法结束。 |

例 3-7 算法中的第 3~5 步使用的是一种自然语言里常用的句型，即“**如果……，就……，否则……**”。在算法设计中，这种句型描述的是“如果**条件**成立，则执行**算法分支** 1，否则执行**算法分支** 2”，这种类型的算法结构被称为选择结构或分支结构。条件、算法分支 1 和算法分支 2 是选择结构中的 3 个要素。

C++语言提供了 2 种**选择语句**（selection statement）句型来描述选择结构的算法，分别是 if-else 语句和 switch-case 语句。

### 3.3.1　if-else 语句

C++语法：if-else 语句

```
if (表达式)
{   语句 1   }
else
{   语句 2   }
```

语法说明：

- **表达式**指定一个判断条件。该表达式结果应为布尔类型，例如关系表达式或逻辑表达式。非布尔类型的表达式结果将被自动转换，0 转为 false，非 0 值转为 true。
- **语句** 1 是描述算法分支 1 的 C++语句序列，即条件成立时执行的语句。
- **语句** 2 是描述算法分支 2 的 C++语句序列，即条件不成立时执行的语句。如果条件不成立时不需要执行什么处理，则省略 else 和{ 语句 2 }。
- 语句 1、语句 2 可能是包含多条 C++语句的序列，此时必须用一对大括号{ }将它们括起来。如果只包含一条 C++语句，则大括号可以省略。
- **计算机执行**该语句时，首先计算表达式（即判断条件），若结果为 true（条件成立），则执行语句 1；否则，执行 else 后面的语句 2。

使用 if-else 语句将例 3-7 的求倒数算法编写成 C++程序，见例 3-8。

**例 3-8　实现求倒数算法的 C++程序（if-else 语句）**

```
#include <iostream>
using namespace std;

int main( )
{
    double x;           // 定义一个 double 型变量 x
    cin >> x;           // 从键盘输入变量 x 的值

    if (x != 0)         // 判断条件“x 不等于 0”是否成立
    {                   // 如果条件成立，则执行以下求倒数的代码
        double y;       // 再定义一个 double 型变量 y，用于保存 x 的倒数
        y = 1 / x;      // 求 x 的倒数，结果赋值给 y
        cout << y;      // 显示 y 的值，即 x 的倒数
    }
    else                // 否则执行以下代码
        cout << "0 的倒数没有意义";          // 显示错误信息
                                             // else 分支只有一条语句，可省略大括号
    return 0;
}
```

用一对大括号{ }括起来的语句序列称为**复合语句**。例 3-8 中的第 10~14 行就是一条复合语句。在语法上 C++语言将复合语句当做一条语句。有了复合语句，上述 if-else 语句的语法定义可以省略掉大括号。

```
if (表达式)
{   语句 1   }
else
{   语句 2   }
```

可简写为：

```
if (表达式)
    语句 1
else
    语句 2
```

其中，语句 1、语句 2 可以是单条语句，也可以是由大括号括起来的复合语句。C++语言还有一种特殊的**空语句**，即仅由“;”构成的语句。计算机执行空语句时不做任何处理。如无特别说明，本书后续语法定义中的术语“语句”都将包括复合语句和空语句。

例 3-9 给出一个判断年份是否闰年的 C++程序。平年的 2 月只有 28 天。公历闰年指的是当年的 2 月有 29 天。粗略地说是四年一闰，而准确判断闰年的条件是：年份能被 4 整除并且不能被 100 整除，或者年份能被 400 整除。该条件比较复杂，例 3-9 代码第 9 行通过取余运算和关系运算来描述“整除”条件，再通过逻辑运算来描述“并且”和“或者”这样的组合条件。

**例 3-9　判断年份是否闰年的 C++程序**

```
1   #include <iostream>
2   using namespace std;
3
4   int main( )
5   {
6       int year;                   // 定义一个 int 型变量 year
7       cin >> year;                // 从键盘输入一个年份，保存到变量 year 中
8
9       if ( (year%4 == 0 && year%100 != 0) || year%400 == 0 )   // 指定是否闰年的判断条件
10          cout << year << “是闰年” << endl;        // 条件成立则该年份是闰年
11      else
12          cout << year << “不是闰年” << endl;  // 否则该年份不是闰年
13      return 0;
14  }
```

例 3-10 给出了使用 if-else 语句求解符号函数的 C++程序。符号函数的定义如下：

$$\text{sgn}(x) = \begin{cases} 1 & (x > 0) \\ 0 & (x = 0) \\ -1 & (x < 0) \end{cases}$$

**例 3-10　求符号函数 sgn(*x*)的 C++程序**

```
1  #include <iostream>
2  using namespace std;
3
4  int main( )
5  {
6      float x;                    // 定义一个 float 型变量 x
7      cin >> x;                   // 从键盘输入变量 x 的值
8
9      int sgn;                    // 定义一个 int 型变量 sgn，用于保存结果
10     if (x == 0)                 // 首先将 x 分为等于 0 和不等于 0 两种情况
11         sgn = 0;                //  x 等于 0 时，sgn = 0
12     else               // 在 x 不等于 0 的情况下，再进一步区分 x>0 和 x<0 这两种情况
13     {
14         if (x > 0)   sgn = 1;   //  x 大于 0 时，sgn = 1
15         else   sgn = –1;        //  x 小于 0 时，sgn = –1
16     }
17
18     cout << sgn << endl;        // 显示变量 sgn 的值，即符号函数的结果
19     return 0;
20 }
```

例 3-10 中的代码第 14~15 行是在 if-else 语句中**嵌套**的另一个 if-else 语句。if-else 语句可以多层嵌套。多层嵌套时应注意：每个 else 自动和上面最近的没有 else 的 if 配对。如果 if-else 配对错误，执行程序得到的结果通常也是错误的。为保险起见，上层 if-else 语句应添加大括号将下层的 if-else 括起来，例如上述代码第 13 和 16 行的大括号。

编写 C++源程序时，良好的**书写格式**对程序的阅读理解非常有帮助。例如例 3-10 中的代码第 14~15 行，大括号内部语句的缩进就是一种很好的书写格式。缩进可以体现语句的层次。添加注释、适当的空行和空格等也都是好的书写格式。另外，代码第 10~11 行可以写在同一行：

```
if (x == 0)   sgn = 0;
```

多条比较短的语句可以写在一行，一条长的语句也可以写成多行。程序的书写格式主要是为方便程序员阅读，不会影响程序语法的正确性。

例 3-10 中，求解符号函数的算法实际上是一种**多分支结构**算法。描述多分支结构算法可以改用另一种特殊的 if-else 句型，即 if-else if 语句，如例 3-11 所示。

**例 3-11　求符号函数 sgn(*x*)的 C++程序（if-else if 语句）**

```
#include <iostream>
using namespace std;

int main( )
{
    float x;                    // 定义一个 float 型变量 x
    cin >> x;                   // 从键盘输入变量 x 的值

```

```
    int sgn;                    // 定义一个 int 型变量 sgn，用于保存结果
    if (x == 0)    sgn = 0;     // 首先检查 x 等于 0 的情况
    else if (x > 0)    sgn = 1; // 否则，再检查 x 大于 0 的情况
    else    sgn = –1;           // 最后剩下的就是 x 小于 0 的情况

    cout << sgn << endl;        // 显示变量 sgn 的值，即符号函数的结果
    return 0;
}
```

C++语法：if-else if 语句

```
if (表达式 1)   语句 1
else if (表达式 2)   语句 2
…
else if (表达式 n)   语句 n
else   语句 n+1
```

语法说明：

- **表达式** 1~n 分别是依次指定的判断条件。表达式的结果应为布尔类型，如关系表达式或逻辑表达式。非布尔类型的表达式结果将被自动转换，0 转为 false，非 0 值转为 true。
- **语句** 1~n 分别对应条件成立时执行的语句，可以是单条语句、复合语句或空语句。
- **语句** n+1 是所有条件都不成立时执行的语句，可以是单条语句、复合语句。如果所有条件都不成立时不需要执行什么处理，即空语句，则省略 else 和语句 n+1。
- **计算机执行**该语句时，首先计算表达式 1，若为 true 则执行语句 1；否则继续计算表达式 2，…，直到表达式 n；如果所有条件都不成立则执行 else 后面的语句 n+1。计算机只会执行语句 1~n+1 中的一条。

if-else if 语句适用于描述多分支结构算法。例 3-12 给出了另一个应用 if-else if 语句的程序实例。该程序的功能是输入表示星期几的数值（1~7），显示其对应的英文单词。

例 3-12　显示星期几英文单词的 C++程序（if-else if 语句）

```
#include <iostream>
using namespace std;

int main( )
{
    int x;              // 定义一个 int 型变量 x
    cin >> x;           // 从键盘输入一个表示星期几的数值（1~7），保存到变量 x 中

    // 下列 if-else if 语句根据 x 的值显示对应星期几的英文单词
    if (x = = 1)    cout << "Monday" << endl;
    else if (x = = 2)    cout << "Tuesday" << endl;
    else if (x = = 3)    cout << "Wednesday" << endl;
    else if (x = = 4)    cout << "Thursday" << endl;
    else if (x = = 5)    cout << "Friday" << endl;
    else if (x = = 6)    cout << "Saturday" << endl;
    else if (x = = 7)    cout << "Sunday" << endl;
```

```
        else    cout << "Input Error" << endl;    // 输入的数值不在 1~7 范围之内，提示错误
        return 0;
}
```

下面再介绍一下 C++语言中的**条件运算符**“? :”。在程序设计中，“如果**条件**成立，则执行**算法分支** 1，否则执行**算法分支** 2”是一种常用的算法结构。例如，比较两个变量 a、b 的大小，将较大值赋值给 c，用 if-else 语句编写的示例代码如下：

```
int a = 5, b = 10, c;
if (a > b)    c = a;
else    c = b;
```

C++语言提供了一种特殊的条件运算符，可以实现这样比较简单的 if-else 结构。

**C++语法：条件运算符“? :”**

**表达式 ？ 表达式 1 : 表达式 2**

语法说明：

- 用条件运算符将三个表达式连接起来，这样所构成的长表达式称为**条件表达式**。其中的**表达式**指定一个判断条件，该表达式结果应为布尔类型。非布尔类型的表达式结果将被自动转换，0 转为 false，非 0 值转为 true。
- 如果**表达式**的结果为 true，则计算**表达式 1**，将其结果作为整个条件表达式的结果；否则计算**表达式 2**，将其结果作为整个条件表达式的结果。
- 条件运算符为三目运算符，优先级为 13，结合性为从右到左。

举例：int a = 5, b = 10, c;

```
a>b ? a : b                 // 这是一个条件表达式，其结果等于 10，数据类型为 int 型
cout << (a>b ? a : b);      // 显示条件表达式的结果
c = a>b ? a : b;            // 将条件表达式的结果赋值给变量 c
```

## 3.3.2　switch-case 语句

多分支结构算法中有这样一类特殊的算法：某一表达式的结果可分为若干种情况，每种情况要求执行一个算法分支。例 3-13 具体描述了这样一类特殊的多分支结构算法。

**例 3-13　一类特殊的多分支结构算法**

1 计算某个表达式，判断其结果属于下列哪种情况。
2 情况 1：执行算法分支 1，执行结束转到 7。
3 情况 2：执行算法分支 2，执行结束转到 7。
4 ……
5 情况 n：执行算法分支 n，执行结束转到 7。
6 否则属于其他情况：执行算法分支 n+1，执行结束转到 7。
7 算法结束。

C++语言提供的 switch-case 语句可描述这类特殊的多分支结构算法。

**C++语法：switch-case 语句**

```
switch (表达式)
{
  case 常量表达式 1: 语句 1
  case 常量表达式 2: 语句 2
  ……
  case 常量表达式 n: 语句 n
  default: 语句 n+1
}
```

语法说明：

- **计算机执行**该语句时，首先计算 switch 后面的表达式，然后将结果依次与各 case 后的常量表达式的结果进行比对。若比对成功，则以比对成功的 case 语句为起点，顺序执行后面的所有语句，直到整个 switch-case 语句结束，或遇到 break 语句时中途结束执行，继续执行 switch-case 语句的下一条语句。如果所有比对都不成功，则将 default 语句作为执行的起点。
- **表达式**的结果应当是整型（即 char、short、int 或 long 型），不能是浮点型。
- **常量表达式** 1~n 分别列出 switch 后“表达式”可能的结果。常量表达式只能是常量，或由常量组成的表达式。各常量表达式的结果不能相同。
- **语句** 1~n 分别对应常量表达式比对成功时应执行的语句序列。通常都在末尾增加一条 break 语句，这样可以宣告算法结束，中途跳出。
- **语句** n+1 是 default 后面的语句，即所有比对都不成功时应执行的语句。default 语句习惯上被放在最后。语句 1~n+1 为复合语句时，大括号也可省略。

switch-case 语句俗称为**开关语句**。可以将例 3-12 显示星期几英文单词的程序改用 switch-case 语句来实现，见例 3-14。

**例 3-14　显示星期几英文单词的 C++程序（switch-case 语句）**

```
#include <iostream>
using namespace std;

int main( )
{
    int x;                  // 定义一个 int 型变量 x
    cin >> x;               // 从键盘输入一个表示星期几的数值（1~7），保存到变量 x 中

    // 下列 switch-case 语句根据 x 的值显示对应星期几的英文单词
    switch ( x )
    {
    case 1:   cout << "Monday" << endl;    break;
    case 2:   cout << "Tuesday" << endl;   break;
    case 3:   cout << "Wednesday" << endl;   break;
    case 4:   cout << "Thursday" << endl;   break;
```

```
    case 5:   cout << "Friday" << endl;   break;
    case 6:   cout << "Saturday" << endl;   break;
    case 7:   cout << "Sunday" << endl;   break;
    default:   cout << "Input Error" << endl;   break;
    }
    // 每个 case 语句显示出对应的英文单词之后，程序功能即已完成
    // 使用 break 语句中途跳出 switch 语句，继续执行其后面的语句（本例中为 return 语句）
    return 0;
}
```

一年有 12 个月，月份有大小。大月份为 31 天，小月份为 30 天。例 3-15 的程序能够显示不同月份的天数。

**例 3-15　显示不同月份天数的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
    int month;              // 定义一个 int 型变量 month
    cin >> month;           // 从键盘输入一个月份（1~12），保存到变量 month 中

    // 下列 switch-case 语句显示不同月份的天数
    switch ( month )
    {
    case 1:    cout << "31 天" << endl;          break;   // 1 月大
    case 2:    cout << "28 或 29 天" << endl;    break;   // 2 月是一个特殊的小月份
    case 3:    cout << "31 天" << endl;          break;   // 3 月大
    case 4:    cout << "30 天" << endl;          break;   // 4 月小
    case 5:    cout << "31 天" << endl;          break;   // 5 月大
    case 6:    cout << "30 天" << endl;          break;   // 6 月小
    case 7:    cout << "31 天" << endl;          break;   // 7 月大
    case 8:    cout << "31 天" << endl;          break;   // 8 月大
    case 9:    cout << "30 天" << endl;          break;   // 9 月小
    case 10:   cout << "31 天" << endl;          break;   // 10 月大
    case 11:   cout << "30 天" << endl;          break;   // 11 月小
    case 12:   cout << "31 天" << endl;          break;   // 12 月大
    default:   cout << "Input Error" << endl;    break;   // 输入错误
    }
    return 0;
}
```

例 3-15 中，所有的大月份都是 31 天，case 1、3、5、7、8、10、12 执行的 cout 语句都是一样的。类似地，所有的小月份 case 4、6、9、11 也是相同的。switch-case 语句中，不同的 case 可以共用语句。通过共用语句，例 3-15 可改写成例 3-16。

**例 3-16　显示不同月份天数的 C++程序（共用语句）**

```
#include <iostream>
using namespace std;

int main( )
{
    int month;  // 定义一个 int 型变量 month
    cin >> month;  // 从键盘输入一个月份（1~12），保存到变量 month 中

    // 下列 switch-case 语句显示不同月份的天数
    switch ( month )
    {
    case 1:    // 1 月大
    case 3:    // 3 月大
    case 5:    // 5 月大
    case 7:    // 7 月大
    case 8:    // 8 月大
    case 10:   // 10 月大：1、3、5、7、8、10 月将和 12 月共用下面显示天数的语句
    case 12:   cout << "31 天" << endl;   break;   // 12 月大
    case 4:    // 4 月小
    case 6:    // 6 月小
    case 9:    // 9 月小：4、6、9 月将和 11 月共用下面显示天数的语句
    case 11:   cout << "30 天" << endl;   break;   // 11 月小
    // 2 月比较特殊，单独使用一条显示天数的语句
    case 2:    cout << "28 或 29 天" << endl;   break;
    default:   cout << "Input Error" << endl;    break;   // 输入错误
    }
    return 0;
}
```

通过例 3-16 可以看出，case 比对的过程实际上是在查找 switch 语句执行的起点。查找到起点后，计算机将从该起点开始，顺序执行后面的所有语句，直到 break 语句中途结束或整个 switch-case 语句结束为止。在 switch-case 语句中，break 语句的作用是中途跳出，转去执行 switch-case 语句后面的下一条语句。

## 本节习题

1. 执行 C++语句：if (1 < 0 || false)　cout << " Hello world! "; 显示器上将显示（　　）。

   A."Hello world!"　　　　B. Hello, world!

   C. Hello world!　　　　D. 什么都未显示

2. 执行下列 C++语句：

```
double x = 0;
if ( x )   cout << 1 / x;
else   cout << x;
```

显示器上将显示（　　）。

A. 0　　B. ∞

C. 显示错误信息　　D. 什么都未显示

3. 执行下列 C++语句：

```
int x = 15;
if (x%2 == 0)    cout << x / 2;
else    cout << x / 2 + 1;
```

显示器上将显示（　　）。

A. 7　　B. 7.5　　C. 8　　D. 8.5

4. 执行下列 C++语句：

```
int x = 2;
switch ( x )
{
case 1:    cout << "One";    break;
case 2:    cout << "Two";    break;
case 3:    cout << "Three";    break;
default:    cout << "Error";    break;
}
```

显示器上将显示（　　）。

A. One　　B. Two　　C. Three　　D. Error

5. 执行下列 C++语句：

```
int x = 1;
switch ( x+1 )
{
case 1:    cout << "One";
case 2:    cout << "Two";
case 3:    cout << "Three";
default:    cout << "Error";
}
```

显示器上将显示（　　）。

A. One　　B. Two　　C. TwoThree　　D. TwoThreeError

## 3.4　循环语句

有一些算法，在满足特定条件下将重复执行某些操作步骤，这种算法结构称为**循环结构**。例如一个奇数数列：1, 3, 5, 7, 9, …，如需计算数列前 $N$ 项的累加和，该如何设计算法呢？通过数学方法，我们可以将这个求累加和问题描述为：$\sum_{n=1}^{N} 2n-1$。这种描述方法本身就体现了一种求解累加和的算法思想：首先引入一个表示数列项的变量 $n$，第 $n$ 项可表示

为 $2n-1$；求解前 $N$ 项累加和的过程是一个重复累加的过程，累加起点是 $n=1$，累加条件是 $n \leqslant N$；每次累加所做的操作就是累加第 $n$ 项（当前项）的值 $2n-1$，然后将 $n$ 加 1，准备下一次累加；重复该累加操作，直到累加条件 $n \leqslant N$ 不成立。上述求解累加和的算法就是一种循环结构算法。每次循环后，算法所引入的变量 $n$ 都会发生变化（加 1），然后通过比较 $n$ 的值来控制是否继续循环（即检查条件 $n \leqslant N$ 是否成立）。像变量 $n$ 这样用于控制循环次数的变量被称为**循环变量**。

循环结构用自然语言描述就是这样一种句型，即：“**如果……，就重复做……，否则停止**”。在算法设计中，这种句型描述的是“如果**条件**成立，则重复执行**循环体**，否则**结束循环**”。一个循环结构由 4 个要素构成，它们分别是：循环变量、循环变量的初始值、循环条件和循环体。例 3-17 具体描述了上述求解累加和的算法。

| 例 3-17　算法举例：求解奇数数列前 *N* 项的累加和 | |
|---|---|
| 1 | 首先定义一个 int 型变量 N，从键盘输入 N 的值。 |
| 2 | 定义一个循环变量 n（初始值为 1），表示当前数列项的序号。 |
| 3 | 再定义一个 int 型变量 sum（初始值为 0），用于保存累加的结果。 |
| 4 | 开始循环：如果循环条件 n≤N 成立，则转到 5 做累加操作，否则转到 7 结束循环。 |
| 5 | 将当前项的值 2n−1 累加到 sum 上：sum +=2n – 1。 |
| 6 | 将 n 加 1，准备下一次累加，转到 4 继续循环。步骤 5~6 是被重复执行的循环体。 |
| 7 | 循环结束后，显示 sum 的值，此时 sum 中的值就是数列前 N 项的累加和。 |
| 8 | 算法结束。 |

C++语言提供了 3 种**循环语句**（iteration statement）句型来描述循环结构的算法，它们分别是 while 语句、do-while 语句和 for 语句。

### 3.4.1　while 语句

C++语法：while 语句

```
while (条件表达式)
        循环体语句
```

语法说明：

- **条件表达式**指定一个循环条件，其结果应为布尔类型，例如关系表达式或逻辑表达式。非布尔类型的表达式结果将被自动转换，0 转为 false，非 0 值转为 true。
- **循环体语句**是描述循环体的 C++语句，即条件成立时被重复执行的语句。如果循环体包含多条语句，则必须用大括号{ }括起来。
- 如果循环条件一开始就不成立，则循环体一次也不执行。循环体中应包含使循环条件趋向于 false 的语句，否则循环条件一直为 ture，循环体将无休止地执行，俗称为**死循环**。
- 计算机执行该语句时，首先计算条件表达式（即循环条件），若结果为 true（条件成立），则执行循环体语句，然后再返回继续判断循环条件是否成立。重复这个过程，直到循环条件不成立时结束循环。

使用 while 语句将例 3-17 的求累加和算法编写成 C++程序，见例 3-18。

**例 3-18　求解奇数数列前 *N* 项累加和的 C++程序（while 语句）**

```
1   #include <iostream>
2   using namespace std;
3
4   int main( )
5   {
6       int N;                      // 定义一个 int 型变量 N
7       cin >> N;                   // 从键盘输入变量 N 的值
8
9       // 定义循环变量 n（初始值为 1）和保存累加结果的变量 sum（初始值为 0）
10      int n = 1, sum = 0;
11      while (n <= N)              // 用小括号将循环条件 n<=N 括起来
12      {
13          sum += 2*n - 1;         // 将当前项的值 2n–1 累加到 sum 上
14          n++;   // 将 n 加 1，准备下一次累加。该语句使得循环条件 n<=N 趋向于 false
15                 // 执行完循环体最后一条语句之后，转到第 11 行，重新判断循环条件
16      }
17      // 循环结束后，继续执行 while 语句后面的语句
18      cout << sum << endl;        // 显示变量 sum 的值，即前 N 项的累加和
19      return 0;
20  }
```

## 3.4.2　do-while 语句

while 语句所描述的循环结构是“如果**条件**成立，则重复执行**循环体**，否则**结束循环**”。该循环结构的一个变形是“**先执行**一次循环体，**再判断条件**。如果条件成立则重复执行循环体，否则结束循环”。C++语言使用 do-while 语句来描述这种循环结构。

C++语法：do-while 语句

```
do
    循环体语句
while (条件表达式) ;
```

语法说明：

- **条件表达式**指定一个循环条件，其结果应为布尔类型，例如关系表达式或逻辑表达式。非布尔类型的表达式结果将被自动转换，0 转为 false，非 0 值转为 true。
- **循环体语句**是描述循环体的 C++语句。如果循环体包含多条语句，则必须用大括号{ }括起来。
- 循环条件被放在循环体语句的后面，即先执行循环体，再判断条件，即无论循环条件是否成立，循环体至少执行一次。循环体中应包含使循环条件趋向于 false 的语句，否则将造成死循环。
- 计算机执行该语句时，首先执行一次循环体，然后再计算条件表达式（即循环条件），若结果为 true（条件成立），则重复执行循环体语句，否则结束循环。

使用 do-while 语句也可以实现例 3-17 的求累加和算法，见例 3-19。

**例 3-19　求解奇数数列前 *N* 项累加和的 C++程序（do-while 语句）**

```
#include <iostream>
using namespace std;

int main( )
{
    int N;                        // 定义一个 int 型变量 N
    cin >> N;                     // 从键盘输入变量 N 的值

    // 定义循环变量 n（初始值为 1）和保存累加结果的变量 sum（初始值为 0）
    int n = 1, sum = 0;
    do                            // 先执行循环体
    {
        sum += 2*n - 1;           // 将当前项的值 2n—1 累加到 sum 上
        n++;                      // 将 n 加 1，准备下一次累加
    } while (n <= N) ;  // 后判断循环条件。如果条件成立则重复执行循环体，否则结束循环
                            // 循环结束后，继续执行 do-while 语句后面的语句
    cout << sum << endl;          // 显示变量 sum 的值，即前 N 项的累加和
    return 0;
}
```

while 语句是先判断条件，再决定是否执行循环体，循环体可能一次也不执行；而 do-while 语句是先执行循环体，再判断条件，循环体至少执行一次。通常情况下，这个细微的差别对算法结果没有影响，两种语句可以互相替换使用。但当一开始的初始条件就不成立时，while 语句和 do-while 语句的执行结果会有差异。例如，在求累加和的程序中，如果从键盘输入的 *N* 是 0，例 3-18 使用 while 语句的计算结果正确（0，没有累加），而例 3-19 使用 do-while 语句的计算结果是错误的（1，累加了一次）。

### 3.4.3　for 语句

C++语言中，循环结构“如果**条件**成立，则重复执行**循环体**，否则**结束循环**”可以用 while 语句描述，也可以用 for 语句描述。使用 for 语句来描述循环结构算法，形式更加紧凑。

C++语法：for 语句

```
for (表达式 1; 表达式 2; 表达式 3)
    循环体语句
```

语法说明：

- **表达式 1** 只在正式循环前执行一次，通常用于为循环算法中的变量赋初始值。
- **表达式 2** 指定一个循环条件。每次循环时，先计算该表达式，如果为 true 则执行下面的循环体语句，否则结束循环。

- **表达式** 3 在每次循环体执行结束之后都被执行一次，主要用于修改循环条件中的某些变量，使循环条件趋向于 false。
- **循环体语句**是描述循环体的 C++语句。
- 计算机执行该语句时，首先计算表达式 1（通常是赋初始值），然后进入循环：计算表达式 2（即循环条件），若结果为 true 则执行循环体语句；执行完循环体语句后，计算表达式 3（通常用于修改循环条件中的某些变量）；然后再返回表达式 2 重新判断条件。重复上述过程，直到表达式 2 的结果为 false（即循环条件不成立）时，结束循环。

for 语句是 3 种循环语句中最简洁的语句。使用 for 语句替换例 3-18 中的 while 语句，同样可以实现求累加和的算法，见例 3-20。

**例 3-20　求解奇数数列前 *N* 项累加和的 C++程序（for 语句）**

```
#include <iostream>
using namespace std;

int main( )
{
    int N;                          // 定义一个 int 型变量 N
    cin >> N;                       // 从键盘输入变量 N 的值

    // 定义循环变量 n 和保存累加结果的变量 sum（初始值为 0）
    int n, sum = 0;
    for (n = 1; n <= N; n++) // 集中使用 3 个表达式来指定 n 的初始值为 1、循环条件为 n<=N，
                             // 并在每次循环后将循环变量 n 的值加 1，使循环条件趋向于 false
    {
        sum += 2*n-1;    // 循环体被简化了，原来的 n++语句被放入到 for 语句里面
    }  // 循环体只有一条语句，此时这对大括号可以省略
    // 循环结束后，继续执行 for 语句后面的语句
    cout << sum << endl;   // 显示变量 sum 的值，即前 N 项的累加和
    return 0;
}
```

结合 for 语句，我们介绍一下 C++语言中的**逗号运算符**“,”及其应用。逗号运算符“,”可以将多个表达式连接起来，构成一个长的**逗号表达式**。

C++语法：逗号运算符“,”

| **表达式 1，表达式 2，……，表达式 n** |
|---|

语法说明：

- 计算机从左到右依次计算**表达式** 1~**表达式** n，并将**最后一个表达式**（即表达式 n）的结果作为整个逗号表达式的结果。
- 逗号运算符的优先级为 15（最低），结合性为从左到右。

```
举例：int a, b, c;
    a = 5, b = 10, c = a+b              // 计算完该表达式后，a 的值为 5，b 的值为 10，c 的值为 15
    // 整个逗号表达式的结果等于最后一个表达式 c=a+b 的结果：数值为 15，类型为 int 型
    cout << (a = 5, b = 10, c = a+b);   // 显示逗号表达式的结果，显示结果为：15
```

应用逗号表达式可以改写例 3-20 中的 for 语句，如例 3-21 所示。

**例 3-21　求解奇数数列前 *N* 项累加和的 C++程序（更加紧凑的 for 语句）**

```
1   #include <iostream>
2   using namespace std;
3   
4   int main( )
5   {
6       int N;                              // 定义一个 int 型变量 N
7       cin >> N;                           // 从键盘输入变量 N 的值
8   
9       int n, sum;                         // 将 n 和 sum 的初始化都放入 for 语句内部
10      for (n=1, sum=0; n <= N; n++)       // 形式更加紧凑的 for 语句
11          sum += 2*n－1;
12      cout << sum << endl;                // 显示变量 sum 的值，即前 N 项的累加和
13      return 0;
14  }
```

例 3-21 代码第 9~11 行可进一步简化为：

```
for (int n=1, sum=0; n <= N; sum +=2*n-1, n++) ;
```

在该 for 语句中，表达式 1 定义变量并初始化，表达式 2 指定循环条件，表达式 3 完成了循环体的功能，真正的循环体变成了一条空语句。根据语法规则，这条 for 语句的执行结果与简化前是一样的，但初学者阅读起来会比较费解。适当运用逗号运算符可以简化程序，但过度使用会给程序的阅读理解造成困难。

## 3.4.4　break 语句和 continue 语句

计算机通常是按照语句的书写顺序来执行 C++程序的，而某些语句会造成执行顺序的跳转。例如下面的示例代码：

```
int a, b, c;
cin >> a >> b;
if (a > b)
   c = a;
else
   c = b;
cout << c << endl;
```

其中，选择语句“if (a > b)”会造成执行顺序的跳转。当计算机执行该选择语句时，如果条件不成立则跳过语句“c = a;”，转去执行 else 后面的语句“c = b;”。这时，程序没有按

照书写顺序执行，而是出现了跳转。造成程序执行顺序跳转的语句被统称为**控制语句**。选择语句和循环语句都属于控制语句。本节再介绍另外两个常用的控制语句。

### 1. break 语句

我们已经知道，用 break 语句能够中途跳出 switch 语句，转去执行 switch 语句后面的下一条语句。使用 break 语句也能够中途跳出循环，转去执行该循环语句的下一条语句。例 3-22 是一个求圆面积的 C++程序。

**例 3-22　计算圆面积的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
      double r;                                // 定义一个变量 r 来存放圆的半径
      cin >> r;                                // 从键盘输入圆的半径
      cout << 3.14*r*r << endl;                // 显示圆面积
      return 0;
}
```

该程序一次只能求一个圆的面积，计算完就退出了。如果需要计算多个圆的面积，但不希望每次都重新启动程序，该怎么设计算法呢？设计思路如下：使用循环结构重复输入半径并显示圆面积的过程。但循环多少次应当由用户决定，该如何设定循环条件呢？可以将循环条件先设定为 true（即死循环），然后根据用户从键盘输入的半径来决定是否结束循环。如果用户输入的半径为正数，则计算圆面积，否则跳出循环，程序结束。因为半径为 0 或负数是没有意义的，可用作结束循环的条件。具体程序代码如例 3-23 所示。

**例 3-23　计算多个圆面积的 C++程序（break 语句应用实例）**

```
1   #include <iostream>
2   using namespace std;
3
4   int main( )
5   {
6         double r;                            // 定义一个变量 r 来存放圆的半径
7         while (true)                         // 死循环
8         {
9               cin >> r;                      // 从键盘输入圆的半径
10              if (r <= 0)    break;          // 如果用户输入的半径小于或等于 0，则跳出循环
11              cout << 3.14*r*r << endl;      // 显示圆面积
12        }
13        // 使用 break 语句中途跳出 while 语句后，继续执行下面的语句（本例中为 return 语句）
14        return 0;
15  }
```

例 3-23 代码第 10 行是在循环语句中嵌套的一个 if-else 语句。C++语言中，所有的选择语句和循环语句之间都可以互相嵌套。循环语句的相互嵌套，即一个循环语句中再包含

另一个循环语句，这被称为是**多重循环**。例 3-24 给出一个生成乘法表的 C++程序，其中使用了二重循环。程序运行结果如图 3-3 所示。

**例 3-24　生成乘法表的 C++程序（多重循环）**

```
#include <iostream>
using namespace std;

int main( )
{
    int x, y;                          // 定义 2 个循环变量 x 和 y
    for (x = 1; x <= 9; x++)           // 第一重循环，x 从 1 到 9，共 9 行
    {
        for (y = 1; y <= x; y++)       // 第二重循环，y 从 1 到 x。第 x 行有 x 个乘法
            cout << y << "×" << x << "=" << x*y << "   ";  // 规范显示格式，例如：2×3=6
        cout << endl;                  // 换一行，再显示后续的内容
    }
    return 0;
}
```

```
"C:\Users\Thinkpad\Desktop\test\Debug\test.exe"
1×1=1
1×2=2  2×2=4
1×3=3  2×3=6  3×3=9
1×4=4  2×4=8  3×4=12  4×4=16
1×5=5  2×5=10  3×5=15  4×5=20  5×5=25
1×6=6  2×6=12  3×6=18  4×6=24  5×6=30  6×6=36
1×7=7  2×7=14  3×7=21  4×7=28  5×7=35  6×7=42  7×7=49
1×8=8  2×8=16  3×8=24  4×8=32  5×8=40  6×8=48  7×8=56  8×8=64
1×9=9  2×9=18  3×9=27  4×9=36  5×9=45  6×9=54  7×9=63  8×9=72  9×9=81

Press any key to continue
```

图 3-3　由 C++程序生成的乘法表

注意，在多重循环中若使用 break 语句，只能跳出它所在的本层循环。另外需要注意的是，break 语句只能在 switch-case 语句和循环语句中使用，否则编译器会提示语法错误。

### 2．continue 语句

continue 语句的作用是结束本次循环，中途返回，继续下一次循环。例 3-25 给出一个 continue 语句的应用实例。

**例 3-25　显示 1~50 之间所有能被 3 整除的数（continue 语句应用实例）**

```
#include <iostream>
using namespace std;
```

```

int main( )
{
    int n;                          // 定义一个循环变量 n
    for (n = 1; n <= 50; n++)       // 从 1 到 50 进行循环
    {
        if (n%3 != 0)   continue;   // 如果 n 不能被 3 整除，则执行 continue 语句
                                    // 其作用是结束本次循环，中途返回，继续检查下一个数
        cout << n << ", ";   // 未中途返回的数是能被 3 整除的数，显示并用逗号隔开
    }
    return 0;
}
```

在循环语句中，continue 语句与 break 语句的区别是：continue 只结束本次循环，而 break 结束的是整个循环。另外，continue 语句只能在循环语句中使用，break 还可以在 switch-case 语句中使用。

## 本节习题

1. 执行下列 C++语句：

```
int x = 5, y = 0;
while (x > 0)
{
    y += 2;   x--;
}
```

执行结束后，x 和 y 的值分别为（　　）。

A. 5, 0　　B. 0, 5　　C. 5, 10　　D. 0, 10

2. 执行下列 C++语句：

```
int x = 5, y = 0;
do {
    y += 2;   x--;
} while (x > 0);
```

执行结束后，x 和 y 的值分别为（　　）。

A. 5, 0　　B. 0, 5　　C. 5, 10　　D. 0, 10

3. 执行下列 C++语句：

```
int y = 0;
for (int x = 5; x > 0; x--)
    y += 2;
```

执行结束后，x 和 y 的值分别为（　　）。

A. 5, 0　　B. 0, 5　　C. 5, 10　　D. 0, 10

4. 执行下列 C++语句：

```
for (int x=0, y = 0; x < 5; y += 2, x++) ;
```

执行结束后，x 和 y 的值分别为（　　）。

A. 5, 0　　B. 0, 5　　C. 5, 10　　D. 0, 10

5. 执行下列 C++语句：

```
int x = 5, y = 0;
while (x > 0)
{
    y += 2;   x--;
    if (x%4 == 0)   break;
}
```

执行结束后，x 和 y 的值分别为（　　）。

A. 5, 0　　B. 0, 10　　C. 4, 2　　D. 4, 4

6. 循环体至少被执行一次的循环语句是（　　）。

A. while 循环　　B. do-while 循环　　C. for 循环　　D. 任意一种循环

7. 执行下列 C++语句：

```
int x = 0;
while (x < 3)
{
    cout << "*";   x++;
}
```

显示器将显示（　　）。

A. *　　B. **

C. ***　　D. ******…，持续显示星号

8. 执行下列 C++语句：

```
int x = 0;
while (x < 3)
    cout << "*";   x++;
```

显示器将显示（　　）。

A. *　　B. **

C. ***　　D. ******…，持续显示星号

## 3.5 算法设计与评价

程序设计中的算法应当具有实用价值并且是可实现的，因此我们所设计的算法应当具备以下 5 个特性。

（1）**有穷性**。一个算法必须保证其执行步骤是有限的。

（2）**确定性**。算法的每个步骤必须有明确的定义，不能含糊不清，或有二义性。

（3）**有效性**。算法的每个步骤应该能被计算机执行，并得到有效的结果。

（4）**输入**。算法可以没有输入，也可以有一个或多个输入，输入是算法处理的原始

数据。

（5）**输出**。算法至少要有一个输出，输出是算法处理的结果。

针对同一程序设计任务，不同程序员可以有不同的算法设计，进而编写出不同的程序。如何评价一个算法的优劣呢？除了正确性之外，计算复杂度和内存占用量是评价算法性能的两个基本标准。

## 3.5.1　计算复杂度

评价算法通常使用**计算复杂度**来描述算法的运算次数。例如，已定义变量“double x=5;”，则执行语句：

```
x *= x;
```

该算法做 1 次乘法，其算法复杂度记为 $O(1)$。执行下列循环语句：

```
for (int n = 1; n <= N; n++)
    x *= x;
```

该算法做 $N$ 次乘法，其算法复杂度与 $N$ 的大小有关系，记为 $O(N)$。执行下列二重循环语句：

```
for (int m = 1; m <= N; m++)
    for (int n = 1; n <= N; n++)
        x *= x;
```

该算法做 $N \times N$ 次乘法，其算法复杂度与 $N^2$ 的大小有关系，记为 $O(N^2)$。上述三种计算复杂度存在如下的大小关系：

$$O(1) \leqslant O(N) \leqslant O(N^2)$$

算法的计算复杂度决定了计算机执行该算法所需的时间。完成相同的功能，计算复杂度越小，算法的执行速度越快，算法就越好。

例如，给定一个整数 $x$，设计一个判断 $x$ 是否素数的算法。算法的设计思路如下：依次检查 $x$ 能否被 2 到 $x-1$ 之间的数整除。如果有一个数能整除，则 $x$ 不是素数，否则就是素数。该算法是一个循环结构，例 3-26 给出了实现算法的 C++程序。

**例 3-26　判断素数的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
    int x;   // 定义一个 int 型变量 x
    cin >> x;   // 从键盘输入变量 x 的值

    // 定义 bool 型变量 yes_no，用来表示 x 是否为素数
    bool yes_no = true;                  // 先假定 x 是素数，因此将 yes_no 的初始值设为 true
    for (int n = 2; n < x; n++)          // 判断 x 是否为素数的循环，从 2 到 x-1 进行循环
```

```
    {
        if (x%n == 0)                        // 如果 x 能被某个 n 整除，则 x 就不是素数
        {
            yes_no = false;   break;         // 将 yes_no 置为 false，跳出循环
        }
        // 如果 x 不能被 n 整除，则 yes_no 保持为 true 不变，继续检查下一个数
    }
    // 循环结束后，如果 yes_no 一直保持为 true 不变，则 x 是素数
    if (yes_no == true)   cout << "是素数" << endl;
    else   cout << "不是素数" << endl;
    return 0;
}
```

如果 $x$ 是素数，上述算法将循环 $x$−2 次，计算复杂度为 $O(N)$。可以对该算法进行优化，只检查 $x$ 能否被 2 到 $x$/2 之间的数整除就可以了。因为 $x$ 肯定不能被 $x$/2+1 到 $x$−1 之间的数整除，没必要检查，这样可以将算法的计算复杂度降为 $O(N/2)$。实际上，该算法还能进一步优化，即将检查范围从[2, $x$/2]再缩小到[2, $\sqrt{x}$ ]，这样算法的计算复杂度会更低。降低算法复杂度，优化算法设计，可以有效提高程序运行的速度。

### 3.5.2　内存占用量

内存占用量用于评估计算机执行算法时所需内存空间的大小。完成相同的功能，内存占用量越少，算法就越好。例 3-27 给出一个计算 3 个同学某门课程平均成绩的 C++程序。

**例 3-27　计算平均成绩的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
    double score1, score2, score3;         // 定义 3 个 double 型变量，分别保存 3 个同学的成绩
    double sum;                            // 定义一个 double 型变量，保存 3 个成绩的总和
    cin >> score1 >> score2 >> score3;     // 从键盘输入 3 个同学的成绩
    sum = score1 + score2 + score3;        // 计算成绩总和
    cout << sum/3 << endl;                 // 计算并显示平均成绩
    return 0;
}
```

例 3-27 定义了 4 个 double 型变量，每个 double 型变量占用 8 个字节的内存，共 32 个字节。通过合理选择数据类型、减少变量个数等手段可以减少内存占用，优化算法设计。例如，选用 float 类型完全可以满足成绩的精度要求，每个 float 型变量只占 4 个字节；每个同学的成绩在累加之后就不需要再保存了，修改算法设计可以让 3 个成绩共用一个变量。例 3-28 给出了算法优化后的 C++程序，其中只定义了 2 个 float 型变量（共 8 个字节），所占用的内存仅为例 3-27 的四分之一。

**例 3-28　计算平均成绩的 C++程序（算法优化之后）**

```
#include <iostream>
using namespace std;

int main( )
{
    float score;                    // 只定义一个保存成绩的 float 型变量
    float sum = 0;                  // 定义一个 float 型变量（初始值为 0），保存 3 个成绩的总和
    cin >> score;   sum += score;         // 从键盘输入第 1 个同学的成绩，累加到 sum
    cin >> score;   sum += score;         // 从键盘输入第 2 个同学的成绩，累加到 sum
    cin >> score;   sum += score;         // 从键盘输入第 3 个同学的成绩，累加到 sum
    cout << sum/3 << endl;                // 计算并显示平均成绩
    return 0;
}
```

## 3.5.3　算法设计举例

程序员面对一个具体的程序设计任务，需要通过分析提炼，设计出能被计算机执行的算法，然后用计算机语言描述该算法。这就是编写程序的过程，算法设计是其中的难点。刚开始练习编程时，初学者常常会表现出一脸茫然，无从下手。

编程能力实际上是一个人计算思维能力的反映。现代人的科学思维能力主要体现在实证思维（例如物理和化学）、逻辑思维（例如数学）和计算思维（例如程序设计）三个方面。计算思维能力不是天生的，必须通过后天学习和培养才能获得。初学者学习程序设计时，阅读程序和模仿编程是培养计算思维能力的两个重要途径。读者在阅读教材中示例程序时，应当问自己“这个程序为什么这么编，每条语句的功能是什么，编写的语法是什么？”阅读程序需要“慢阅读”，在阅读过程中仔细体会算法设计的思维方式。阅读程序后可以再独立编写一遍，或者模仿编写另外一个类似的程序，这样可以有效提高自己的编程能力。本节给出几个示例程序，供读者进行阅读和模仿练习时使用。

### 1. 求逆序数

从键盘输入一个整数，编程计算并输出它的逆序数。例如，输入-123，则应输出-321。以下为程序的一个运行示例：

Input: -123<回车键>

Output: -321

任何人都能手工完成上述求逆序数的工作，但如何让计算机完成这样的工作呢？初学者通常是不知道如何设计算法，或者是所设计的算法不完善（例如忘记处理负数）。例 3-29 给出一个求逆序数的 C++示例程序。

**例 3-29　求逆序数的 C++示例程序**

```
#include <iostream>
using namespace std;

```

```
int main( )
{
    int n;
    cout << "Input: ";      cin >> n;        // 提示用户输入并保存到变量 n 中
    cout << "Output: ";                      // 显示一个提示输出结果的信息"Output:"
    // 输入为负数时要先处理其中的负号
    if (n < 0)      // 如果 n 为负数，则先显示负号，然后转成正数再进行处理
    {
        cout << "-";      n = -n;   // 显示负号，然后将 n 转成正数
    }
    int lowest;             // 定义一个保存 n 中最低位（个位）的变量 lowest
    while (n > 0)           // 从个位开始，通过循环依次求 n 中的各个位
    {
        lowest = n % 10;          // 通过取余运算求出 n 的个位
        cout << lowest;
        n /= 10;   // 除以 10 消掉个位，同时其他各位也依次降位，进入下一次循环
        // 当 n 中所有的位都被取出后，n 变为 0，循环条件 n>0 不成立，循环结束
    }
    cout << endl;
    return 0;
}
```

2. 水仙花数

所谓“水仙花数”，是指一个各位数字的立方和等于该数本身的 3 位数。例如 153 就是一个“水仙花数”，因为 $1^3+5^3+3^3=153$。编写程序求出所有的水仙花数。

求水仙花数的算法思路是：对所有 100~999 之间的 3 位数，逐个检查它们是否符合水仙花数的条件（即各位数字的立方和等于该数本身），并将符合条件的数显示出来。该算法首先确定问题的范围，然后对范围内所有可能的情况逐一验证从而求得问题的解，这种求解问题的方法被称为**穷举法**。穷举法是算法设计中一种常用的方法，需要用循环结构来实现。例 3-30 给出一个用穷举法求水仙花数的 C++示例程序。运行该程序，所求出的水仙花数有 153、370、371 和 407。

**例 3-30　求水仙花数的 C++示例程序**

```
#include <iostream>
using namespace std;

int main( )
{
    int a, b, c;                              // 定义 3 个变量分别保存 3 位数的百位、十位和个位
    for (int n=100; n <= 999; n++)   // 通过循环依次检查所有 100 到 999 之间的 3 位数
    {
        a = n / 100;              // 取出数 n 的百位
        b = (n / 10) % 10;        // 取出数 n 的十位
        c = n % 10;               // 取出数 n 的个位
        if (a*a*a + b*b*b + c*c*c == n)          // 检查是否满足水仙花数的条件
            cout << n << " ";                // 显示满足条件的水仙花数（用空格隔开）
```

```
        }
        cout << endl;
        return 0;
}
```

### 3. 求反正切函数

本节最后再给出一个求反正切（arctan）函数的算法设计举例。计算机的 CPU 只能做算术运算、关系运算等基本运算，不能直接求解三角函数、对数等复杂运算，需要基于基本运算来设计求解算法。求解 arctan($x$)算法的设计思路如下。

为了用计算机求解反正切函数，需通过级数展开将其转换为基本的算术运算：

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

求解 arctan($x$)的问题被转换成了一个求级数累加和的问题：

$$\arctan(x) = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{2n+1}, \text{记 } f(n) = \frac{x^{2n+1}}{2n+1}$$

将$(-1)^n$的正负号问题转换为判断 $n$ 的奇偶性问题，偶数项做加法，奇数项做减法。

数列项 $f(n)$的分子和分母也分别构成一个数列，分别记为 $a(n)$和 $b(n)$，则：

$$a(0) = x，a(n+1) = a(n)\times x^2$$

$$b(0) = 1，b(n+1) = b(n) + 2$$

$$f(n) = a(n) / b(n)$$

使用循环结构可以实现求级数累加和的算法。循环必须是有穷的，可以根据精度要求来设计循环条件。例如，将循环条件设为 $f(n)\geqslant 10^{-5}$，即当数列项的值小于 $10^{-5}$ 时结束累加循环。

将上述算法编写成 C++程序，如例 3-31 所示。

**例 3-31　求解 arctan($x$)的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
    double x;
    cin >> x;                    // 从键盘输入正切值 x

    double sum = 0;              // sum 用于保存数列的累加和，初始值为 0
    int n = 0;                   // n 用于保存当前数列项的序号，初始值为 0
    double a = x;                // a 用于保存当前数列项分子的值，初始值等于 x
    double b = 1;                // b 用于保存当前数列项分母的值，初始值等于 1
    double f;                    // f 用于保存当前数列项的值
    do                           // 循环累加
    {
        f = a / b;
        sum = (n%2 == 0) ? sum+f : sum-f;      // 偶数项做加法，奇数项做减法
```

```
        n++;                              // 数列项序号加 1，准备累加下一项
        a *= x*x;   b += 2;               // 计算出下一项的分子和分母
    } while (f >= 1e-5) ;                 // 循环条件为 f(n)≥10^-5
    // 循环结束后，sum 中保存的是反正切函数的结果（以弧度为单位的角度）
    cout << sum * 180 / 3.1415926 << endl;    // 将弧度单位转换成以度为单位，显示结果
    return 0;
}
```

### 学习本章的要点

- 读者要了解，绝大部分复杂算法都可以由三种基本的算法结构来实现。
- 读者要掌握布尔类型及其相关的运算符。
- 读者要通过案例认真体会算法设计的思维方式，并通过阅读程序和模仿编程训练来提高自己的编程能力。
- 读者要能根据算法结构合理选用控制语句，并熟练掌握其编写时的语法规则。

## 3.6 本章习题

1. **阅读程序**。阅读下列 C++程序，说明程序的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
using namespace std;
int main( )
{
    int N;
    cin >> N;
    bool yes_no = true;
    for (int n = 2; n <= N/2; n++)
    {
        if (N % n == 0)
        {
            yes_no = false;    break;
        }
    }
    if (yes_no == true)    cout << "Yes" << endl;
    else    cout << "No" << endl;
    return 0;
}
```

2. **程序改错**。阅读下列 C++程序，并检查其中的语法错误。修改错误，并保证程序的功能不变。

```
#include <iostream>
using namespace std;
int main( )
```

```
{
    int n = 20;                         // 显示 20 以内的奇数
    while (n > 0);                      // while 循环，循环条件为 n>0
    {
        if (n % 2 = 0)                  // 如果 n 为奇数，则显示
            cout << n << endl;
        n++;                            // 检查下一个数是否奇数
    }
    return 0;
}
```

3. **编写程序**。分别用 while 语句、do-while 语句和 for 语句编写一个求阶乘 $N!$的 C++程序。

4. **编写程序**。我国古代《张丘建算经》中有这样一道著名的百鸡问题："鸡翁一，值钱五；鸡母一，值钱三；鸡雏三，值钱一。百钱买百鸡，问鸡翁、母、雏各几何？"这道题的大意是：公鸡每只 5 元，母鸡每只 3 元，小鸡 3 只 1 元。用 100 元买 100 只鸡，问公鸡、母鸡和小鸡各能买多少只？编写一个求解百鸡问题的 C++程序。

# 第4章 数组与文字处理

数据是程序处理的对象。数据要存放在内存中才能被 CPU 读取和处理，处理后的结果也只能保存回内存中。C++语言通过定义变量来申请保存数据所需的内存空间。程序员需为变量命名，变量之间不能重名。每个变量分配一个内存单元，只能保存一个数据。当需要保存并处理大量数据时，C++程序该如何定义变量呢？

问题举例：某一门课程有 100 名同学选修，老师按学号顺序给出了课程成绩单。如何编写一个 C++程序来计算这门课程的平均成绩、最高分和最低分呢？作为程序员，首先要考虑该如何定义变量来存储这 100 名同学的成绩数据，然后再考虑设计什么样的算法来求平均值、最大值和最小值。需要在程序中定义 100 个变量吗？答案当然是否定的，为 100 个变量命名就已经是一件很麻烦的事情了。为了存储这样一组成绩数据，C++语言提供了一种特殊的数据组织形式，称为数组。

**数组**（array）是一组类型相同并按某种次序排列的数据集合，其中的每个数据被称为是数组的一个**元素**（element）。数组元素按排列次序编号，编号为从 0 开始的整数。编号被称为数组元素的**下标**（subscript）。

存储一维方向排列的数据（例如数列）使用一维数组，一维数组有 1 个下标；存储二维方向排列的数据（例如矩阵）使用二维数组，二维数组有 2 个下标，第 1 个为**行**下标，第 2 个为**列**下标……。C++语言可以定义多维数组，但最常用的是一维数组和二维数组。

计算机只能存储和处理数值数据，而**文字处理**程序所处理的对象是字符数据。为了存储和处理字符数据，需要使用某种编码标准将字符转换成数值编码。早期的计算机只能处理英文，ASCII 码就是专门为英文设计的编码标准（详见 1.4.3 节的表 1-5），其中包括 10 个阿拉伯数字、52 个英文字母（含大小写）、33 个其他常用符号，另外还包括 33 个控制字符（称为不可见字符或不可打印字符，例如 Esc 就是一个控制字符，其编码为 27），合计 128 个字符。每个字符分别对应 0~127 之间的一个整数编码，用 7 位二进制来表示即 000 0000 ~ 111 1111。计算机用一个字节（8 位）来存储一个字符编码，最高位未用到，置为 0。

C++语言使用字符类型来存储字符的编码，该编码是一个单字节整数。C++语言将字符类型与单字节整数类型合二为一，统称为**字符类型**（char）。在计算机内部，一个字符实际上就是一个单字节整数。一个字符类型变量可以保存一个字符，而要保存一篇文章（即一组字符）则需要使用**字符数组**。



# 4.1 数组

C++程序中可以定义一个**数组变量**（可简称为数组）来保存一组数据。限制条件就是这组数据的数据类型必须相同，该类型称为数组变量的数据类型。定义数组变量时须指定数组元素的个数。每个数组元素相当于一个普通变量，可以存放一个数据。访问某个数组元素时需通过下标来指明访问的是哪个数组元素。

数组变量中存放的是一个数据集合。对数据集合最常规的处理方法是依次访问集合中的每个元素，将所有元素逐个处理一遍，这种处理方法称为对数据集合的**遍历**。设计遍历算法需要用到循环结构。

## 4.1.1 数组变量的定义与访问

### 1. 定义数组变量

C++语法：定义数组变量

**数据类型　数组变量名[常量表达式 1][常量表达式 2]……;**

语法说明：

- **数据类型**指定了数组变量的数据类型。
- **数组变量名**可简称为**数组名**，需符合标识符的命名规则。
- **常量表达式**指定了多维数组各下标的个数，用中括号“[ ]”括起来。常量表达式可以是单个常量，或是由常量组成的表达式，其结果必须为正整数。
- 定义一维数组需要 1 个常量表达式，用来指定数组元素的个数（称为该数组的长度）；定义二维数组需要 2 个常量表达式，第 1 个指定行数，第 2 个指定列数，数组元素个数=行数×列数……数组元素的个数等于各常量表达式的乘积。

举例：

```
int x[5] ;                  // 定义一个 int 型一维数组变量 x，包含 5 个数组元素
double y[2+3] ;             // 定义一个 double 型一维数组变量 y，包含 5 个数组元素
int z[5][10] ;              // 定义一个 int 型二维数组变量 z，包含 5 行 10 列，共 50 个数组元素
int a, x[5], z[5][10] ;     // 可以将相同类型的数组变量和普通变量放在一条语句中定义
```

当执行数组变量定义语句时，计算机为数组中的所有元素同时分配内存空间。各数组元素的内存单元在内存中是按顺序连续排列的。数组变量所占字节数等于各元素所占字节数的总和。第 2 章表 2-1 列出了 10 种常用基本数据类型占用内存的字节数，C++语言还另外提供一种 sizeof 运算符来自动获取某种数据类型或某个变量所占用的字节数。

C++语法：sizeof 运算符

**sizeof( 数据类型名 )**

或

**sizeof( 表达式 )**

语法说明：

- sizeof( **数据类型名** )的计算结果是所指定数据类型占用的字节数。
- sizeof( **表达式** )的计算结果是括号中表达式结果的类型所占用的字节数。表达式可以是单个变量、常量或数组变量，这时 sizeof 的计算结果就是该变量、常量或数组变量所占用的字节数。

例如，以下是一些由 sizeof 运算符构成的表达式例子。

```
sizeof( int )          // 计算结果是数据类型 int 占用的字节数：4
sizeof( double )       // 计算结果是数据类型 double 占用的字节数：8
sizeof( 2.0 )          // 常量 2.0 默认的数据类型是 double，因此计算结果为 8
sizeof( 2 + 3.5 )      // 表达式 2+3.5 结果的数据类型是 double，因此计算结果为 8
```

假设，已定义普通变量 a 和数组变量 b“int a, b[5] ;”，则：

```
sizeof( a )            // 普通变量 a 是 int 型，因此计算结果为 4
sizeof( b[0] )         // 数组元素 b[0]是 int 型，因此计算结果为 4
sizeof( b )            // 数组变量 b 所占字节数等于所有元素占用字节数的总和，结果为 20
```

数组变量在定义之后，可通过数组名和下标来访问其中的任意一个元素。

### 2. 访问数组元素

#### C++语法：访问数组元素

**数组变量名[下标表达式 1][下标表达式 2]……**

语法说明：

- **数组变量名**指明要访问哪个数组。
- **下标表达式**指明所访问数组元素的下标。多维数组要指明所有的下标，分别用中括号“[ ]”括起来。假设下标表达式 n 所对应数组定义中的下标个数为 N，则该表达式的结果应当为 0~N−1 之间的非负整数，0 是该下标的下界，N−1 是该下标的上界。访问数组元素时下标不能越界，否则将出现**越界错误**。编译器检测不出程序中的数组越界错误，但执行时可能会导致不可预知的后果，例如死机等。
- 访问一维数组元素须给出 1 个下标表达式；访问二维数组元素须给出 2 个下标表达式，第 1 个是行下标，第 2 个是列下标……。

例如，若定义一维数组变量 x：

```
int x[3];              // 包含 3 个元素，分别为 x[0]、x[1]和 x[2]
```

则可访问该一维数组的各个元素，例如：

```
x[0] = 10 ;            // 将数组变量 x 的第 0 个元素赋值为 10
x[1] = 20 ;            // 将数组变量 x 的第 1 个元素赋值为 20
x[2] = 30 ;            // 将数组变量 x 的第 2 个元素赋值为 30
x[3] = 40 ;            // 越界错误：数组变量 x 的下标 3 超过了其上界 2
```

访问数组元素时，用中括号“[ ]”将下标括起来，这一对中括号被称为**下标运算符**。计算机执行下标运算符，就是根据下标来访问指定数组元素的内存单元。访问包括写入数据或读出数据。

如果定义二维数组变量 y：

```
int y[2][3];                    // 包含 2 行 3 列，共 6 个元素
```

则可访问该二维数组的各个元素，例如：

```
y[0][0] = 10 ;                  // 将数组变量 y 第 0 行第 0 列的元素赋值为 10
y[0][1] = 20 ;                  // 将数组变量 y 第 0 行第 1 列的元素赋值为 20
y[0][2] = 30 ;                  // 将数组变量 y 第 0 行第 2 列的元素赋值为 30

y[1][0] = 40 ;                  // 将数组变量 y 第 1 行第 0 列的元素赋值为 40
y[1][1] = 50 ;                  // 将数组变量 y 第 1 行第 1 列的元素赋值为 50
y[1][2] = 60 ;                  // 将数组变量 y 第 1 行第 2 列的元素赋值为 60

y[2][0] = 70 ;                  // 越界错误：数组变量 y 的行下标 2 越界，行的上界为 1
y[0][3] = 40 ;                  // 越界错误：数组变量 y 的列下标 3 越界，列的上界为 2
```

访问数组元素时可以使用变量或表达式来指定数组元素的下标。例如：

```
int x[3], n = 0 ;
x[n] = 10 ;                     //n 为 0，将数组变量 x 的第 0 个元素赋值为 10
x[n+1] = 20 ;                   //n+1 为 1，将数组变量 x 的第 1 个元素赋值为 20
x[n+2] = 30 ;                   //n+2 为 2，将数组变量 x 的第 2 个元素赋值为 30

x[n+3] = 40 ;                   // 越界错误：n+3 为 3，下标 3 超过了其上界 2
x[n-1] = 40 ;                   // 越界错误：n–1 为–1，下标–1 超过了其下界 0
```

### 3. 数组的整体输入和输出

可以使用 cin 语句从键盘输入所有的数组元素，称为数组的**整体输入**。也可以使用 cout 语句将所有数组元素输出到显示器上，这称为数组的**整体输出**。数组整体输入/输出需要遍历所有的数组元素，可以使用循环结构来实现。例 4-1 给出了对一维数组和二维数组进行整体输入/输出的 C++示例代码。

**例 4-1　数组整体输入/输出的 C++示例代码**

```
#include <iostream>
using namespace std;

int main( )
{
    int m, n;                       // 预先为后面的循环语句定义好循环变量

    // 对一维数组进行整体输入/输出的示例代码
    int x[3];                       // 定义一个 int 型一维数组变量 x，包含 3 个元素
    for (n = 0; n < 3; n++)         // 使用循环结构逐个输入各数组元素
        cin >> x[n];                // 从键盘输入数组元素 x[n]
```

```
    // 注意上述 for 语句的循环条件不能是“n <= 3”，否则最后一次循环时将越界
    for (n = 0; n < 3; n++)              // 使用循环结构逐个输出各数组元素
        cout << x[n] << ", ";            // 显示数组元素 x[n]，用逗号隔开
    cout << endl;                        // 显示结束后换一行

    // 对二维数组进行整体输入/输出的示例代码
    double y[2][3];     // 定义一个 double 型二维数组变量 y，2 行 3 列，共 6 个元素
    for (m = 0; m < 2; m++)              // 使用二重循环结构，m 为行下标循环变量
        for (n = 0; n < 3; n++)          // n 为列下标循环变量。将按先行后列的顺序循环
            cin >> y[m][n];              // 从键盘输入第 m 行第 n 列的数组元素 y[m][n]
                                         // 注意这两个下标不能写成 y[n][m]，否则将会越界
    for (m = 0; m < 2; m++)              // 使用二重循环结构逐个输出各数组元素
    {
        for (n = 0; n < 3; n++)          // m 为行下标循环变量，n 为列下标循环变量
            cout << y[m][n] << "   ";    // 显示数组元素 y[m][n]，用空格隔开
        cout << endl;                    // 每显示完一行数组元素，显示器上也换一行
    }                   // 第一重 for 循环包含 2 条语句，语法上要求用大括号括起来
    return 0;
}
```

#### 4. 数组的初始化

定义一个数组变量，各数组元素都被分配了内存单元，但内存单元存放的是以前遗留下来的数据，这些数据是不确定的。定义数组变量时可以为全部或部分元素赋初始值，即数组的初始化。初始值需用大括号“{ }”括起来。

**1）全部初始化**

为所有数组元素赋初始值，例如：

```
int x[3] = { 2, 4, 6 };                    // 将 3 个元素的初始值依次设为 2、4、6
double y[2][3] = { 1, 3, 5, 2, 4, 6 };     // 将第 0 行 3 个元素的初始值依次设为 1、3、5，
                                           // 第 1 行 3 个元素的初始值分别设为 2、4、6
```

当列出全部数组元素的初始值时，可省略第 1 个下标个数。例如，上述两条定义语句都列出了所有元素的初始值，可简写为：

```
int x[ ] = { 2, 4, 6 };                // 一维数组省略下标个数，将根据初始值数量把下标个数设为 3
double y[ ][3] = { 1, 3, 5, 2, 4, 6 }; // 多维数组只能省略第 1 个下标个数
                                       // 将根据初始值数量把行下标个数设为 2（6÷3=2）
```

上述代码在编译时，由编译器根据初始值数量自动计算被省略的下标个数，这样可以略微减少一点程序员的工作量。

初始化二维数组时，可以按行再用大括号将每一行的初始值分别括起来，例如：

```
double y[2][3] = { {1, 3, 5}, {2, 4, 6} };     // 这种初始化格式更容易阅读
```

**2）部分初始化**

为部分数组元素赋初始值，例如：

```
int x[3] = { 2, 4 };                     // 前两个元素的初始值依次设为 2、4
```

```
                                          // 最后一个元素没有给初始值，自动设为 0
double y[2][3] = { {1, 3}, {2, 4} };      // 每一行的最后一个元素没有给初始值，自动设为 0
double y[2][3] = { {1, 3, 5} };           // 将第 0 行 3 个元素的初始值依次设为 1、3、5
                                          // 第 1 行没有给初始值，3 个元素都自动设为 0
```

定义数组变量时，如果没有初始化，则各数组元素的初始值是以前遗留下来的，是不确定的；如果部分初始化，则未初始化元素的初始值自动设为 0。可以使用如下的 C++代码来检查部分初始化后各数组元素的值：

```
int x[3] = { 2, 4 };              // 定义数值变量 x，部分初始化
cout << x[0] << endl;             // 显示结果：2
cout << x[1] << endl;             // 显示结果：4
cout << x[2] << endl;             // 显示结果：0
cout << x[3] << endl;             // 越界，可能会显示一个随机的数值
cout << x[30000] << endl;         // 严重越界，十有八九会造成死机
```

### 4.1.2　常用的数组处理算法

数组变量中存放的是一个数据集合。处理数据集合的算法，例如求数组元素的总和或平均值，通常都要遍历数组中的所有元素，因此需要使用循环结构来实现。遍历多维数组时需要用到多重循环。

#### 1. 求数组元素的总和、平均值

例 4-2 给出了一个求数组元素总和、平均值的 C++程序实例。

**例 4-2　求数组元素总和、平均值的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
    // float 型二维数组变量 y，定义时被初始化了。求该数组的总和、平均值
    float y[2][3] = { { 0.5, 2.5, 4.5 }, { 1.5, 3.5, 5.5 } };

    int m, n;                     // 预先为下面的循环语句定义好循环变量 m 和 n
    float sum = 0, average;       // 定义变量 sum 保存总和（初始值为 0），average 保存平均值

    for (m = 0; m < 2; m++)       // 使用二重循环求二维数组的累加和，m 为行下标循环变量
        for (n = 0; n < 3; n++)   // n 为列下标循环变量
            sum += y[m][n];       // 将第 m 行第 n 列的数组元素 y[m][n]累加到 sum 上

    average = sum / (2*3);        // 根据总和求平均值
    cout << "总和=" << sum << "平均值=" << average << endl;     // 显示总和、平均值
    return 0;
}
```

### 2. 求数组元素的最大值、最小值

例 4-3 给出了一个求数组元素最大值和最小值的 C++程序实例。

**例 4-3 求数组元素最大值、最小值的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
    // int 型一维数组变量 x，定义时被初始化了。求该数组的最大值和最小值
    int x[6] = { 1, 4, 6, 2, 5, 3 };
    int max, min;   // 定义变量 max 保存最大值，min 保存最小值

    max = x[0];   min = x[0];   // 先假设第 0 个元素就是最大值，也是最小值
    for (int n = 1; n < 6; n++)   // 使用循环结构依次将剩余元素与 max、min 进行比较，
                // 再根据比较结果决定 max 和 min 的值。n 为下标循环变量，从 1 到 5 循环
    {
        if (x[n] > max)   max = x[n];   // 如果 x[n]比 max 大，则这个值是最大值，修改 max
        if (x[n] < min)   min = x[n];   // 如果 x[n]比 min 小，则这个值是最小值，修改 min
    }
    cout << "最大值=" << max << "最小值=" << min << endl;   // 显示最大值、最小值
    return 0;
}
```

### 3. 数组排序

排序是一种常用的数组处理算法，它将数组中原来无序的数据集合按照某种规则进行排序，这样可以提高今后查找的速度。例如，定义一个数组变量 x：

```
int x[6] = { 1, 4, 6, 2, 5, 3 } ;
```

对数组 x 进行排序，得到排序后的结果，如图 4-1 所示。

定义一个数组变量x：int x[6] = { 1, 4, 6, 2, 5, 3 } ;

| | 排序前 | | | 排序后 |
|---|---|---|---|---|
| | … | | | … |
| x[0] | 1 | | x[0] | 1 |
| x[1] | 4 | | x[1] | 2 |
| x[2] | 6 | 按从小到大的 | x[2] | 3 |
| x[3] | 2 | 规则进行排序 | x[3] | 4 |
| x[4] | 5 | | x[4] | 5 |
| x[5] | 3 | | x[5] | 6 |
| | … | | | … |

图 4-1 数组排序内存示意图

设计数组排序算法，最直接的思路是：假设数组变量 $x$ 有 $N$ 个元素，先遍历这 $N$ 个元素，找出值最小的元素，假设为 $x[m]$，交换 $x[m]$和 $x[0]$的值；然后再遍历剩下的 $N$−1 个元素，继续查找值最小的元素，将其与 $x[1]$交换……直到只剩一个元素为止。这种通过选择最小元素并进行交换的排序算法被称为**选择法**。例 4-4 给出一个用选择法进行数组排序的 C++程序。

**例 4-4　用选择法进行数组排序的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
    // int 型一维数组变量 x，定义时被初始化了。对该数组按从小到大的规则进行排序
    int x[6] = { 1, 4, 6, 2, 5, 3 };

    int minNo;                    // 定义变量 minNo 用于记录最小元素的下标
    int m, n;                     // 选择法需要二重循环，预先定义好循环变量
    for (m = 0; m <= 4; m++)      // 第一重循环：下标从 0 到 4 循环，直到只剩一个元素时停止
    {
        minNo = m;     // 找出 x[m]~x[5]中的最小元素，用 minNo 记录最小元素的下标
        for (n = m+1; n <= 5; n++)       // 第二重循环：下标从 m+1 到 5 循环
        {
            if (x[n] < x[minNo])    minNo = n;            // minNo 始终记录最小元素的下标
        }       // 使用这一组大括号便于阅读，此处可以省略
        // 交换 x[m]和 x[minNo]的值
        int temp;          // 为了交换两个元素的值，需定义一个临时变量 temp 用于中转数据
        temp = x[m];    x[m] = x[minNo];    x[minNo] = temp;      // 交换两个元素值的方法
    }

    for (m = 0; m < 6; m++)       // 用循环结构整体输出排序后的数组
       cout << x[m] << ", " ;     // 显示数组元素的值，用逗号隔开
    cout << endl ;
    return 0;
}
```

数组排序还有很多种算法，例如冒泡法、插入法等。有兴趣的读者可以参看与“数据结构”相关的书籍。

**4．矩阵的转置**

转置是矩阵的一种基本运算。程序设计中，使用二维数组来存储矩阵，矩阵转置就是对二维数组来操作。例如矩阵 $a = \begin{bmatrix} 0.5 & 2.5 & 4.5 \\ 1.5 & 3.5 & 5.5 \end{bmatrix}$，其转置矩阵为 $a^T = \begin{bmatrix} 0.5 & 1.5 \\ 2.5 & 3.5 \\ 4.5 & 5.5 \end{bmatrix}$。例 4-5 给出一个求矩阵转置的 C++程序。

**例 4-5 矩阵转置的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
    // double 型二维数组变量 a，定义时被初始化了
    double a[2][3] = { { 0.5, 2.5, 4.5 }, { 1.5, 3.5, 5.5 } };   // 保存 2 行 3 列的矩阵
    // 求上述矩阵的转置，转置后的矩阵为 3 行 2 列
    double aT[3][2];   // 再定义一个 3 行 2 列的数组变量 aT，保存转置后的矩阵

    int m, n;   // 预先为后面的循环语句定义好循环变量
    for (m = 0; m < 3; m++)   // 使用二重循环求矩阵的转置，m 为 aT 的行下标循环变量
        for (n = 0; n < 2; n++)              // n 为 aT 的列下标循环变量
            aT[m][n] = a[n][m];              // 注意：访问 a 时，行下标和列下标做了对调

    for (m = 0; m < 3; m++)                  // 使用二重循环整体输出转置后的数组元素
    {
        for (n = 0; n < 2; n++)              // m 为行下标循环变量，n 为列下标循环变量
            cout << aT[m][n] << "  ";        // 显示数组元素 aT[m][n]，用空格隔开
        cout << endl;                        // 每显示完一行数组元素，显示器上也换一行
    }
    return 0;
}
```

## 本节习题

1. 若有 C++数组定义语句“int a[3];”，则数组 a 中不包括下列哪个元素？（　　）
   A. a[0];　　B. a[1];　　C. a[2];　　D. a[3];
2. 下列一维数组定义语句中，语法错误的是（　　）。
   A. int a[5];　　B. int A[ ] = { 1, 2, 3 };
   C. int A[3–5];　　D. int a[3*5];
3. 下列二维数组定义语句中，语法错误的是（　　）。
   A. int a[5][2];　　B. int a[5, 2];
   C. int a[5][2] = { {1, 2}, { 1 } };　　D. int a[ ][2] = { {1, 2}, {1, 1} };
4. 已定义数组“int a[10];”，下列语句中正确的是（　　）。
   A. a0 = 10;　　B. A[0] = 10;
   C. a[3*3] = 10*10;　　D. a[4*4] = 10*10;
5. 访问数组元素时，数组下标不可以是（　　）。
   A. 整型常量　　B. 整型变量　　C. 整型表达式　　D. 负数
6. 已定义数组“int a[100];”，下列哪条语句能将数组 a 中的所有元素都赋值为 10？（　　）
   A. a[0~99] = 10;
   B. a[0] = a[1] = ⋯ = a[99] = 10;
   C. for (int n = 1; n <= 100; n++)　a[n] = 10;
   D. for (int n = 99; n >= 0; n––)　a[n] = 10;

# 4.2　指针与数组

定义数组变量，计算机将为数组变量中的所有元素同时分配内存空间。各数组元素的内存单元在内存中是按顺序连续排列的。图 4-2 分别显示了一维数组变量 x 和二维数组变量 y 各数组元素在内存中的存储顺序。

定义数组变量后，可通过数组名和下标**直接访问**该数组变量中的任意一个元素。第 2 章 2.7.2 节曾介绍了通过指针变量和指针运算符间接访问所指向变量的方法。如果定义一个指针变量 p，将数组中第 0 个元素的地址（称为数组的**首地址**）赋值给 p，我们称指针变量 p 指向了该数组的第 0 个元素，这时就可以通过指针变量 p 来**间接访问**数组的第 0 个元素。可以修改指针变量 p 中的地址值，使之指向其他数组元素，这样就能间接访问数组中的所有元素。通过指针变量和指针运算可以很方便地遍历数组元素。

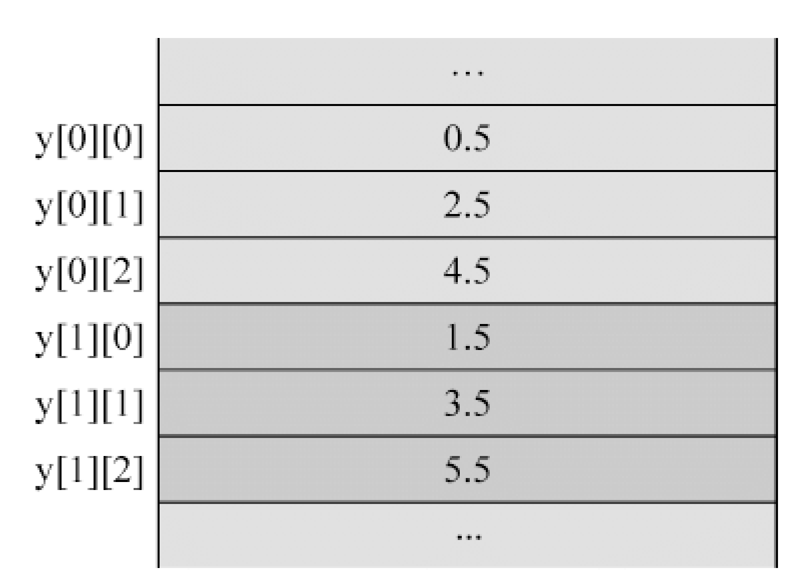
利用数组元素在内存中连续存储的特点，通过加减运算（算术运算）修改地址值可以让指针变量指向不同的数组元素；通过比较地址值的大小（关系运算），可以确定不同数组元素之间的位置次序。指针类型就是地址类型，凡是涉及内存地址的运算统称为**指针运算**。例如上述对内存地址进行的算术运算、关系运算，以及第 2 章 2.7.2 节介绍的取地址和取内容运算等。

int x[6] = { 1, 4, 6, 2, 5, 3 } ;
// 一维数组元素**按下标顺序**连续存储
// 每个 int 型数组元素占 4 个字节

| | |
|---|---|
| | … |
| x[0] | 1 |
| x[1] | 4 |
| x[2] | 6 |
| x[3] | 2 |
| x[4] | 5 |
| x[5] | 3 |
| | ... |

(a) 一维数组

double y[2][3] = { { 0.5, 2.5, 4.5 }, { 1.5, 3.5, 5.5 } } ;
// 二维数组元素**按先行后列的顺序**连续存储
// 每个 double 型数组元素占 8 个字节




(b) 二维数组

图 4-2　数组的存储

## 4.2.1　指针运算

定义和数组类型相同的指针变量，就可以将指针变量指向任意一个数组元素，然后通过指针变量间接访问数组元素。下面的示例代码演示了如何间接访问一维数组元素。

```
int x[6] = { 1, 4, 6, 2, 5, 3 } ;      // 定义一维数组变量 x 并初始化
int *p ;                               // 定义一个和数组变量 x 类型相同的指针变量 p，均为 int 型
p = &x[0] ;                            // 将指针变量 p 指向第 0 个元素
cout << *p ;                           // 通过指针变量 p 间接访问第 0 个元素，显示结果：1
p = &x[1] ;                            // 将指针变量 p 指向第 1 个元素
```

```
cout << *p ;                    // 通过指针变量 p 间接访问第 1 个元素，显示结果：4
……
```

取地址运算符“&”可以取出任意一个数组元素的地址。数组第 0 个元素的地址是数组的首地址。也可以直接用数组名取出一维数组的首地址。例如上述示例代码中的语句：

```
p = &x[0] ;                     // 将指针变量 p 指向第 0 个元素
```

和语句：

```
p = x ;
```

等价。可以将一维数组的数组名理解为是一个表示该数组首地址的符号常量。

间接访问二维数组元素的示例代码如下：

```
double y[2][3] = { { 0.5, 2.5, 4.5 }, { 1.5, 3.5, 5.5 } ;   // 定义二维数组变量 y 并初始化
double *p ;                     // 定义一个和数组变量 y 类型相同的指针变量 p
p = &y[0][0] ;                  // 将指针变量 p 指向第 0 行第 0 列的元素
cout << *p ;                    // 通过指针变量 p 间接访问第 0 行第 0 列的元素，显示结果：0.5
p = &y[1][0] ;                  // 将指针变量 p 指向第 1 行第 0 列的元素
cout << *p ;                    // 通过指针变量 p 间接访问第 1 行第 0 列的元素，显示结果：1.5
…
```

二维数组的每一行都可理解为是一个一维数组。可通过数组名和行下标直接取出每一行的第 0 个元素，例如：

```
p = &y[0][0] ;  与  p = y[0];  等价，都是将指针变量 p 指向第 0 行第 0 列的元素；
p = &y[1][0] ;  与  p = y[1];  等价，都是将指针变量 p 指向第 1 行第 0 列的元素。
```

可以将 y[n]理解为一个表示二维数组 y 第 n 行第 0 列元素地址的常量。

**1．指针的算术运算**

利用数组元素在内存中连续存储的特点，通过加减运算修改地址值可以让指针变量指向不同的数组元素。假设一个指针变量 p 指向某个一维数组变量 x 的第 0 个元素 x[0]，p 和 x 都是 int 型（见图 4-3）。如何修改 p 的地址值使其指向下一个数组元素 x[1]呢？

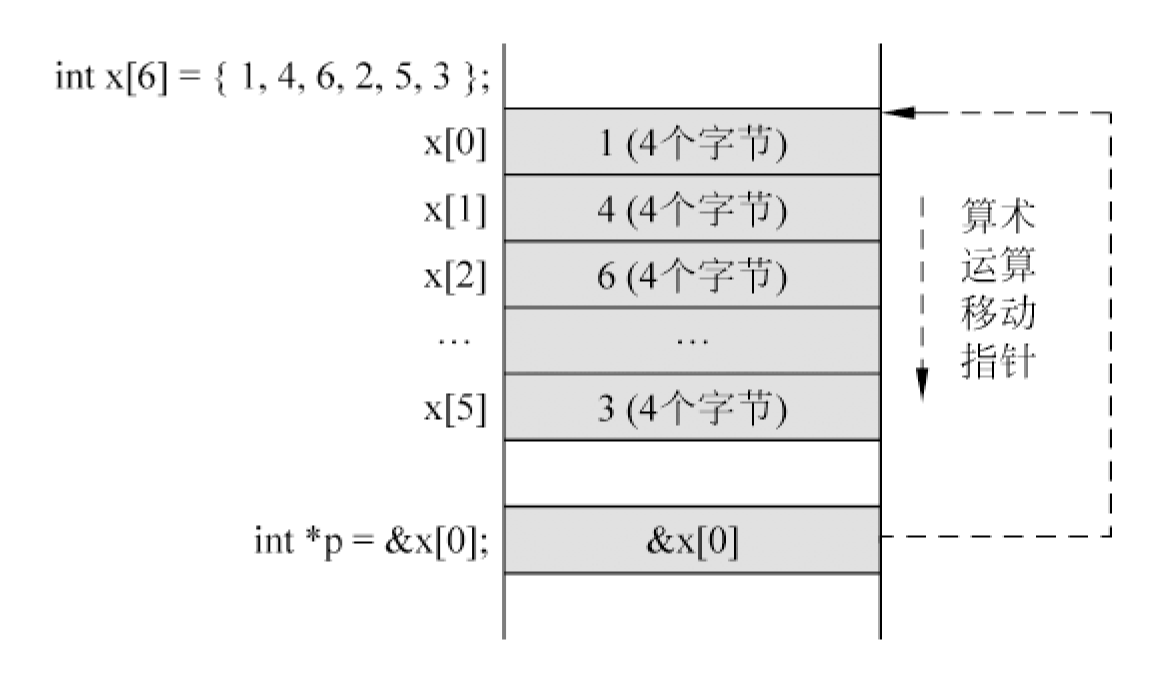



图 4-3　通过算术运算移动指针

每个 int 型数组元素占 4 个字节，相邻数组元素地址的差值为 4。如果想让指针变量 p

指向下一个元素 x[1]，可以将 p 的地址值加 4（例如 p += 4）。而对一个 double 型数组，每个数组元素占 8 个字节，相邻数组元素地址的差值为 8，将指针变量从一个元素移到下一个元素需要将指针变量的地址值加 8。也就是说，程序员使用指针变量遍历数组，在将指针变量从一个元素移到下一个元素时需要考虑到数组的类型，不同类型增加不同的字节数，这样很麻烦。为了方便程序员使用指针变量遍历数组，C++语言对指针算术运算规则给出了如下特殊定义。

（1）**指针类型（非 void）可以与整数进行加减运算**。假设指针变量 p 的数据类型为 T，n 为整数，则表达式“p±n”的结果仍为 T 类型的指针，其地址值等于 p 的地址值±n*sizeof(T)。按照这个运算规则，无论 T 是什么数据类型，只要指针变量 p 指向数组（同为 T 类型）中的某个元素，则 p+n 就是该元素后第 n 个元素的地址，p−n 就是该元素前第 n 个元素的地址。例如，p+1 就是后一个元素的地址，p−1 则是前一个元素的地址。

（2）**两个相同类型（非 void）的指针可以进行减法运算**。假设两个指针变量 p1 和 p2 的数据类型均为 T，则表达式“p1−p2”的结果为 int 型，值等于(p1−p2)/sizeof(T)。按照这个运算规则，如果指针变量 p1 和 p2 分别指向同一数组中的某两个元素，则 p1−p2 的结果就是这两个元素下标的差值。例如，

```
int x[10], *p1=&x[5], *p2=&x[2];
```

则：p1 − p2 等于 3（即等于下标 5 和 2 的差值），&x[5] − &x[2]也等于 3。

（3）**void 型指针不能进行算术运算**。如需对 void 型指针进行算术运算，需要先将其转换成非 void 型指针（例如 int 型），然后再进行运算。**注**：C++语言没有定义 void 类型占多少字节，sizeof( void )是语法错误。

例 4-6 给出一个指针算术运算的 C++演示程序。

**例 4-6　指针算术运算的 C++演示程序**

```
#include <iostream>
using namespace std;

int main( )
{
      int x[6] = { 1, 4, 6, 2, 5, 3 };   // 定义 int 型一维数组变量 x，定义时被初始化了
      int *p, *p1, *p2;                  // 定义 3 个 int 型指针变量，类型与数组变量 x 相同

      p = &x[0];                         // 将指针变量 p 指向第 0 个元素
      for (int n = 0; n < 6; n++)        // 使用循环结构遍历显示数组 x
            cout << *(p+n) << ", " ;     // 通过与整数 n 的算术运算依次访问各数组元素

      int d;                             // 定义一个 int 型变量 d
      p1 = p + 1;                        // 将指针变量 p1 指向第 1 个元素
      p2 = p + 4;                        // 将指针变量 p2 指向第 4 个元素
      d = p2 - p1;   // 指针变量之间的差值等于其所指向数组元素下标之间的差值，即 4–1
      cout << d << endl;                 // 显示结果为 3

      d = &x[4] - &x[1];                 // 数组元素地址之间的差值等于其下标之间的差值，即 4–1
      cout << d << endl;                 // 显示结果也为 3
      return 0;
}
```

### 2. 指针的关系运算

使用关系运算符可以比较两个地址值的大小。如果两个指针变量指向同一数组中的元素，则通过比较它们的大小可以确定其所指向数组元素之间的位置次序。地址值小的元素在前，地址值大的元素在后。如果相等，则说明两个指针变量指向了数组的同一个元素。例 4-7 给出一个指针关系运算的 C++演示程序。

**例 4-7　指针关系运算的 C++演示程序**

```
#include <iostream>
using namespace std;

int main( )
{
    int x[6] = { 1, 4, 6, 2, 5, 3 };        // 定义 int 型一维数组变量 x，定义时被初始化了
    int *p;       // 定义一个 int 型指针变量 p，类型与数组变量 x 相同

    for (p = &x[0]; p <= &x[5]; p++)  // 使用循环遍历显示数组，循环条件采用了指针关系运算
        cout << *p << ", " ;      // 通过指针变量 p 间接访问数组元素，每次循环后执行 p++
    cout << endl;
    return 0;
}
```

### 3. 指针的取内容运算和下标运算

指针运算符“*”也称作取内容运算符，它按照指针变量所保存的地址访问所指向的内存单元。例如，定义一个一维数组变量 x 和一个指针变量 p：

int x[6] = { 1, 4, 6, 2, 5, 3 }, *p = x ;

其中，指针变量 p 指向数组 x 的首地址（即第 0 个元素的地址），这时可以通过指针运算符访问数组的各个元素。具体的访问形式如下：

*p、*(p+1)、*(p+2)、*(p+3)、*(p+4)、*(p+5)

其中，*(p+n)表示所访问的是数组的第 n 个元素。

指针变量 p 也可以通过下标运算符“[ ]”访问数组的各个元素。C++语法规定：“p[n]”与“*(p+n)”等价，其中 p 为指针变量，n 为非负整数。因此，若指针变量 p 指向了数组变量 x 的首地址，则可以通过下标运算符访问各数组元素。具体的访问形式是：

p[0]、p[1]、p[2]、p[3]、p[4]、p[5]

前面章节中，在通过数组名访问时我们一直使用下标来指定访问哪个数组元素。例如访问数组 x 的各数组元素，我们用：

x[0]、x[1]、x[2]、x[3]、x[4]、x[5]

这是一种下标运算符的访问形式。在 C++语言中，一维数组的数组名可理解为是一个表示数组首地址的符号常量。因此，通过数组名访问数组元素时也可以使用如下的指针运算符

形式：

*x、*(x+1)、*(x+2)、*(x+3)、*(x+4)、*(x+5)

无论采用哪种访问形式，本质上它们都是一种通过内存地址间接访问的形式，访问效果也是一样的。实际应用中到底该选择哪种访问形式呢？下面给出一个参考例子。假设有甲乙两位程序员，程序员甲定义一个数组 data，并用 cin 指令输入其中的数据。其示意代码如下：

```
double data[10];                 // 程序员甲定义一个数组 data
for (int m = 0; m < 10; m++)     // 整体输入数组的数据
    cin >> data[m];              // 通常，程序员甲以下标形式访问自己的数组元素
```

程序员甲采用下标形式来访问自己的数组元素，其原因有两个：一是书写简单，二是可以明确表明 data 是一个数组，易于他人阅读理解。

程序员乙可以共享数组 data 中的数据，例如整体输出其中的数据元素。访问他人定义的数组时，程序员乙需要定义一个指针变量来指向该数组的首地址，然后通过指针变量间接访问各数组元素。其示意代码如下：

```
double *pData = data;            // 程序员乙定义指针变量 pData 指向数组 data 的首地址
for (int n = 0; n < 10; n++)     // 整体输出数组中的数据
    cout << *(pData + n);        // 程序员乙以指针形式访问 data 中的数组元素
```

程序员乙也可以采用下标形式访问 data 中的数据元素，例如：

```
cout << pData[n];                // 程序员乙以下标形式访问 data 中的数组元素
```

上述两种访问形式中，指针形式能更明确地表明 pData 是一个指针变量。

## 4.2.2　动态内存分配

程序定义数组变量可以保存大量相同类型的数据。例如本章开头提到的求平均成绩问题，可以定义一个数组变量来保存 100 名同学的成绩。但不同课程学生的人数不一样，定义数组变量时元素个数该定义为多少呢？这需要程序员在设计程序时先预估学生的人数。假设定义一个包含 100 个元素的数组变量，但是当实际学生人数多于 100 人时，数组所申请的内存空间不足；而学生人数少于 100 人时，又会造成内存空间的浪费。

本节介绍 C++语言提供的另一种内存分配方法——**动态内存分配**。动态内存分配就是在程序运行时，根据实际需要**按需分配**内存，使用完之后再**及时释放**。内存的动态分配、访问及释放都需要通过指针变量才能实现。C++语言使用 new 和 delete 运算符进行单个变量或数组变量的动态内存分配与释放，但其使用语法存在一些区别。下面就分别来介绍它们。

### 1. 单个变量的动态分配与释放

通常，C++语言通过定义变量语句申请内存空间，然后通过变量名访问所分配的内存单元。例如：

```
int x;                  // 定义一个 int 型变量 x，定义时指定数据类型和变量名
```

```
                           // 计算机执行该语句将为变量 x 分配一个 4 字节的内存单元
x = 10;                    // 通过变量名 x 访问所分配的内存单元，向其中写入数据 10
cout << x;                 // 通过变量名 x 访问所分配的内存单元，从中读出数据 10 并显示出来
```

可以看出，变量就是内存中的某个内存单元，变量名实际上是其内存单元的名字。动态分配内存时，程序员使用 new 运算符向操作系统申请分配内存，如分配成功则返回所分配内存单元的首地址。动态分配的内存单元没有名字，只有地址。程序员需要预先定义一个指针变量来保存动态分配的内存地址，然后通过指针变量间接访问内存单元。访问结束后，应使用 delete 运算符及时释放内存，所释放的内存单元被交还给操作系统。

**C++语法：单个变量的动态分配与释放**

**指针变量名** = new **数据类型(初始值)** ;

delete **指针变量名**;

语法说明：

- **数据类型**指定动态分配变量的数据类型。
- **初始值**指定所分配内存单元的初始值（用小括号括起来），即变量的初始化。如果不需要初始化，则可以省略“(初始值)”。
- 计算机执行 new 运算符时将按照数据类型指定的字节数分配内存空间并初始化，然后返回所分配内存单元的首地址。应当通过赋值语句将该地址保存到一个预先定义好的同类型**指针变量**中。
- 计算机执行 delete 运算符时将按照**指针变量**中的地址释放指定的内存单元。

举例：动态分配一个 int 型变量。

```
int *p ;                   // 为了动态分配一个 int 型变量，需预先定义好一个 int 型指针变量
p = new int ;              // 使用 new 运算符动态分配一个 int 型变量，该变量没有变量名
                           // 将所分配内存单元的首地址赋值给指针变量 p
*p = 10 ;                  // 通过指针变量 p 间接访问所分配的内存单元，向其中写入数据 10
cout << *p ;               // 通过指针变量 p 间接访问所分配的内存单元，读出数据 10 并显示出来
delete p;                  // 使用结束后，用 delete 运算符释放之前所分配的内存单元
```

上述前 3 条语句可简化为如下的一条语句：

```
int *p = new int(10) ;     // 动态分配变量时进行初始化
```

动态分配方法可以让程序员更主动、更直接地管理内存，根据需要分配尽可能少的内存，同时尽早释放以减少内存的占用时间。

### 2. 一维数组的动态分配与释放

**C++语法：一维数组的动态分配与释放**

**指针变量名** = new **数据类型[表达式]** ;

delete [ ]**指针变量名**;

语法说明：

- **数据类型**指定动态分配数组变量的数据类型。
- **表达式**指定一维数组的元素个数，用中括号“[ ]”括起来。表达式可以是单个常量、变量或是

一个整数表达式，其结果必须为正整数。

- 计算机执行 **new** 运算符时将按照数据类型和元素个数分配相应字节的内存空间，然后返回所分配内存单元的首地址。应当通过赋值语句将该首地址保存到一个预先定义好的同类型**指针变量**中。**注**：动态分配的数组变量不能初始化。
- 计算机执行 **delete** 运算符时将按照**指针变量**中的地址释放指定的内存单元，“[ ]”表示所释放的内存空间是一个数组，其中包含多个内存单元，应同时释放。释放动态分配的数组时必须在指针变量名前加“[ ]”。

举例：动态分配一个 int 型数组变量。

```
int *p = new int[5] ;          // 动态分配一个 int 型一维数组变量，包含 5 个数组元素
*(p+1) = 10 ;                  // 通过指针运算符访问第 1 个元素，向其中写入数据 10
                               // 也可以通过下标运算符访问第 1 个元素：p[1] = 10;
cout << *(p+1) ;               // 通过指针运算符访问第 1 个元素，读出数据 10 并显示出来
                               // 也可以通过下标运算符访问第 1 个元素：cout << p[1];
…
delete [ ]p;                   // 使用结束后，用 delete 运算符释放该数组变量所分配的内存空间
```

例 4-8 给出一个显示 Fibonacci（斐波那契）数列前 $N$ 项的 C++程序。Fibonacci 数列的定义如下：

$$F(0) = 0, F(1) = 1,$$
$$F(n) = F(n-1) + F(n-2) \quad n \geqslant 2$$

例 4-8 通过键盘输入 $N$，然后定义一个包含 $N$ 个元素的数组变量来保存数列的前 $N$ 项。本例中不能使用如下的语句来定义数组 $F$：

```
int F[N];                      // 语法错误
```

用上述形式定义数组 $F$ 时，$N$ 必须是常量，这意味着程序员在编写程序时就必须确定其数值。而本例中，$N$ 的数值是在程序运行时由键盘输入的，编写程序时无法确定，因此只能采用动态分配的方法来定义数组变量 $F$。

**例 4-8　显示 Fibonacci 数列前 *N* 项的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
    int N;                         // 定义一个 int 型变量 N
    cin >> N;                      // 从键盘输入要显示的数列项个数，保存到变量 N 中

    int *F = new int[ N ];         // 动态分配包含 N 个元素的数组，用于保存数列的前 N 项
    F[0] = 0;   F[1] = 1;          // 指定数列前两项的数值

    int n;                         // 为循环语句定义好循环变量 n
    for (n = 2; n < N; n++)        // 使用循环结构计算剩余的数列项
        F[n] = F[n-1] + F[n-2]; // 每一项等于其前两项之和
```

```

    for (n = 0; n < N; n++)          // 使用循环结构遍历显示数组
    {
        cout << F[n] << ", " ;     // 各数列项用逗号隔开
        if ((n+1) % 5 == 0)   cout << endl;   // 一行显示 5 项，即每 5 项换一行显示
    }

    delete [ ]F;                     // 数组使用结束后，释放其内存空间
    return 0;
}
```

### 4.2.3 指针数组

可以定义一个指针类型的数组，其每个元素都是指针变量，这就是**指针数组**。例如：

```
int *pi[3] ;              // 定义 int 型指针数组 pi，每个数组元素都是 int 型指针变量
double *pd[3] ;           // 定义 double 型指针数组 pd，每个数组元素都是 double 型指针变量
```

指针数组可应用于二维数组的动态分配。二维数组的每一行都可以理解为是一个一维数组。依次为每一行动态分配一个一维数组，并用指针数组来保存各行的首地址，这样就实现了二维数组的动态分配。例 4-9 给出一个动态分配二维数组的 C++程序。

**例 4-9　动态分配一个二维数组的 C++程序**

```
#include <iostream>
using namespace std;

int main( )
{
    // 一个 3 行 5 列的二维数组，可以理解为由 3 个一维数组组成
    int *p[3];   // 预先定义一个指针数组 p，用于保存 3 个一维数组的首地址
                 // 指针数组的元素个数应等于二维数组的行数（即 3）

    int m;
    for (m = 0; m < 3; m++)     // 使用循环结构，为每行动态分配一个一维数组
        p[m] = new int[5] ;     // 动态分配一维数组，元素个数应等于二维数组的列数 5
                                // 将所分配内存空间的首地址保存到对应行的指针数组元素中

    // 根据需要访问二维数组的各个元素：p[0][0], …, p[2][4]，此处代码省略

    // 使用结束后应当释放内存
    for (m = 0; m < 3; m++)   // 使用循环结构，逐个释放每一行动态分配的一维数组
        delete [ ]p[m] ;
    return 0;
}
```

指针数组中的每个元素都是一个指针变量。如果使用取地址运算符对指针数组元素进行取地址运算，所取出的地址就是指针变量的地址，即指针的指针，这种指针被称为**二级指针**。可定义保存二级指针的变量，定义时在变量名之间加二级指针声明符“**”。例如：

```
int x = 10 ;                    // 定义一个 int 型变量 x，初始值为 10
int *p = &x ;                   // 定义一个 int 型指针变量 p，初始化指向变量 x
int **pp = &p ;                 // 定义一个 int 型二级指针变量 pp，初始化指向指针变量 p
```

使用二级指针变量，可以将例 4-9 代码第 7 行的定义指针数组语句：

```
int *p[3] ;
```

改为动态分配的形式：

```
int **p = new int *[3] ;
```

其中，声明符“**”表示指针变量 p 是一个二级指针。理论上，C++语言还可以定义更多级的指针变量，但没有太多的实际意义。

### 本节习题

1. 定义一个指向一维数组 x 首地址的指针变量 p，下列语句中错误的是（　　）。
   A. int x[5], p = x[0];　　B. int x[5], *p = x;
   C. int x[5], *p = &x[0];　　D. int x[5], *p = &x[2]−2;
2. 已有定义语句“int a[10], x, *pa = a;”，若要把数组 a 中下标为 3 的元素值赋给 x，则下列语句中不正确的是（　　）。
   A. x = a[3];　　B. x = *(a+3);　　C. x = pa[3];　　D. x = *pa + 3;
3. 执行 C++语句“int x[5], *p = x;　p += 2;”，则指针变量 p 指向数组 x 的哪个元素？（　　）
   A. x[0]　　B. x[1]　　C. x[2]　　D. x[3]
4. 动态分配包含 20 个元素的 int 型数组，下列语句中正确的是（　　）。
   A. int *p = new int[20];　　B. int *p = new int(20);
   C. int *p = new [20];　　D. int p = new int[20];
5. 执行动态分配内存语句“int *p = new int[10];”，释放该内存应使用下列哪个语句？（　　）
   A. delete p;　　B. delete *p;　　C. delete p[10];　　D. delete [ ]p;
6. 语句“int *pi[5];”定义了一个什么数组？（　　）
   A. 一维 int 型数组　　B. 一维 int 型指针数组
   C. 二维 int 型数组　　D. 二维 int 型指针数组

## 4.3 字符类型

计算机只能存储和处理数值形式的数据。为了进行文字处理，计算机需要将文字字符转换成数值编码，这样文字处理问题就转换成了数值计算问题。字符编码需要遵循统一的标准，编码标准要考虑以下两个方面的内容。

（1）确定有哪些字符，即字符集（character set）。

（2）确定每个字符的编码值。

早期的计算机只考虑英文处理，因此由美国国家标准学会 ANSI 制定的 ASCII 码字符集只收录了阿拉伯数字、英文字母、常用符号和控制字符等英文处理需要用到的字符，合计 128 个。分别用 0~127 表示这 128 个字符（参见 1.4.3 节表 1-5 的 ASCII 码表），将十进制的 0~127 转换成二进制就是 000 0000 ~ 111 1111。可以用一个字节（8 位）来存储一个字符编码。最高位未用到，置为 0。ASCII 编码标准具有以下特点：

- 阿拉伯数字 0~9 按从小到大的顺序连续编码。
- 英文字母按字母顺序连续编码。
- 大小写字母分别编码，小写字母的码值比大写字母大 32。
- 0 表示一种特殊字符，称为空字符。

C++语言使用字符类型来存储字符的编码，该编码是一个单字节整数。C++语言将字符类型与单字节整数类型合二为一，统称为 char 型。一个 char 型变量可以保存一个数值较小的整数，例如：

```
char ch;                  // 定义一个 char 型变量 ch
ch = 77;                  // 在变量 ch 中保存整数 77
```

char 型变量只占一个字节，比 short 和 int 型少，可以减少内存的占用。char 类型的数值范围是−128~127，unsigned char 类型的数值范围是 0~255。char 型变量只能保存数值较小的整数，否则会丢失数据（即数据溢出）。

更多时候，char 型变量是用于保存字符的。一个 char 型变量可以保存一个字符，所保存的是该字符的 ASCII 码值。

### 4.3.1 字符型常量

字符型常量是指某个特定的字符，用单引号括起来，例如'a'、'A'、'?'、'@' 等。将字符常量赋值给字符变量的含义就是在该变量中保存这个特定的字符，例如：

```
char ch;
ch = 'M';   // 赋值语句：在字符变量 ch 中保存字符 M
```

实际上，上述赋值语句在 ch 中保存的是字符 M 的 ASCII 码值（77），该语句等价于：

```
ch = 77;
```

在 ASCII 编码字符集中还有 33 个不可见的控制字符，它们无法用上述形式书写，例如空字符（0）、响铃（7）和 Esc（27）等。可以用十六进制或八进制的 ASCII 码值来书写这些不可见字符常量。例如，Esc 的 ASCII 码值为 27，其十六进制为 1B，八进制为 33，则 Esc 的字符常量可写成：

```
'\x1B'   或   '\33'
```

其中，有“x”表示十六进制，无“x”表示八进制。这种书写形式被称为**转义字符**，其中的反斜杠“\”是转义字符的标记。为了方便程序员记忆和使用，C++语言还预定义了一些常用的转义字符常量，见表 4-1。

可见字符也可以使用转义的形式来书写。例如字符'M'的 ASCII 码值为 77，其十六进制为 4D，八进制为 115，因此'M'、'\x4D'或'\115'这三种写法是等价的，都表示字符常量'M'。

表 4-1　常用转义字符常量

| 转义字符 | 含　义 | 注　释 |
|---|---|---|
| \0 | 空字符 | ASCII 码值：0 |
| \a | 响铃 | ASCII 码值：7 |
| \b | 退格 | ASCII 码值：8 |
| \t | 水平制表符（Tab 键） | ASCII 码值：9 |
| \n | 换行 | ASCII 码值：10 |
| \v | 垂直制表符（打印机有效） | ASCII 码值：11 |
| \r | 回车 | ASCII 码值：13 |
| \' | 单引号 | 被赋予了特殊含义，需转义恢复其原来含义 |
| \" | 双引号 | 被赋予了特殊含义，需转义恢复其原来含义 |
| \\ | 反斜杠 | 被赋予了特殊含义，需转义恢复其原来含义 |

## 4.3.2 字符型运算

可以对字符型数据进行算术运算。运算时，将字符的 ASCII 码值作为整数参与运算。例 4-10 给出一个字符型算术运算的 C++演示程序。

**例 4-10　字符型算术运算的 C++演示程序**

```
#include <iostream>
using namespace std;

int main( )
{
    char ch = 'A' ;   // 定义一个 char 型变量 ch，定义时初始化为字符'A'，其 ASCII 码值为 65

    for (int n = 1; n <= 26; n++)      // 使用循环结构显示 26 个大写英文字母
    {
        cout << ch ;  // 根据 ch 中保存的 ASCII 码值，将其所对应的字符显示出来
                      // 只要是 char 型数据，cout 显示的不是 ASCII 码值，而是其对应的字符
        ch++ ;        // 将 ch 中保存的 ASCII 码值加 1，即变成了下一个字母
    }
    cout << '\n' ;    // 显示换行的转义字符'\n'即可实现换行显示，等价于：cout << endl;
    return 0;
}
```

可以对字符型数据进行关系运算。运算时，将字符的 ASCII 码值作为整数进行比较。例 4-11 给出一个字符型关系运算的 C++演示程序。

**例 4-11　字符型关系运算的 C++演示程序**

```
#include <iostream>
using namespace std;

```

```
int main( )
{
    char ch ;                          // 定义一个 char 型变量 ch
    cin >> ch ;  // 从键盘输入一个字符，即单击任一个按键，然后按回车键

    if (ch >= '0' && ch <= '9')        // 关系运算：判断是不是阿拉伯数字
        cout << "阿拉伯数字键" << endl;
    else if (ch >= 'A' && ch <= 'Z')   // 关系运算：判断是不是大写英文字母
        cout << "大写英文字母键" << endl;
    else if (ch >= 'a' && ch <= 'z')   // 关系运算：判断是不是小写英文字母
        cout << "小写英文字母键" << endl;
    else  // 上述条件都不满足，统一显示为“其他按键”
        cout << "其他按键" << endl;

    // 如果想显示 ch 中字符的 ASCII 码值，必须将 ch 转换成其他整数类型
    cout << "该字符的 ASCII 码值= " << (int)ch << endl ;    // 显示字符的 ASCII 码值
    return 0;
}
```

## 本节习题

1. 字母 A 的 ASCII 码是 65（十进制）。下列哪种表示字母 A 的字符常量是错误的？（　　）

A. 'A'　　B. '\65'　　C. '\101'　　D. '\x41'

2. 控制字符“换行”的 ASCII 码值为 10。下列让显示器换行的语句中哪条是错误的？（　　）

A. cout << endl;　　B. cout << '\n';　　C. cout << '\12';　　D. cout << 10;

3. C++语言表达式“'A' >= 'Z' && '0' <= '1'”的结果是（　　）。

A. 'A'　　B. '0'　　C. true　　D. false

4. 字母 A 的 ASCII 码是 65（十进制）。C++语言表达式“5 > 'A'”的数据类型和值分别是（　　）。

A. int，5　　B. char，'A'　　C. bool，true　　D. bool，false

5. 字母 A 的 ASCII 码是 65（十进制）。C++语言表达式“'A' + 1”的数据类型和值分别是（　　）。

A. char，'A'　　B. char，'B'　　C. int，65　　D. int，66

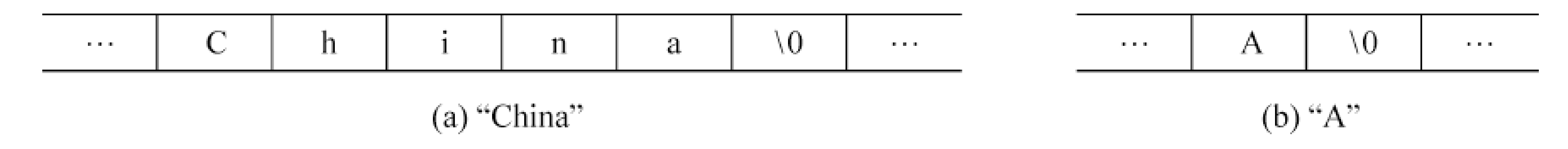
# 4.4 字符数组与文字处理

一个字符类型变量可以保存一个字符，而要保存一篇文章（即一组字符）则需要使用字符类型的数组变量，简称为**字符数组**。一个单词、一句话或一篇文章等都是一种由字符组成的序列，称为**字符串**（string）。C++语言使用字符数组来保存字符串。对字符串的处理主要包括复制、连接、插入和删除等。

## 4.4.1　字符串常量

用双引号括起来的字符序列被称为**字符串常量**，例如“China”、“John”、“Hello world!”和“Please input a number.”等。

计算机会为 C++程序中的字符串常量分配内存空间，并在末尾自动添加空字符'\0'（ASCII 码为 0）作为结束标记。存储一个字符串常量所需内存空间的字节数=字符个数+1。例如，字符串常量“China”的字符个数为 5，再加上 1 个空字符，共占用 6 个字节，如图 4-4(a)所示。用双引号括起来的单个字符也是字符串常量，也会被添加空字符，例如“A”将占用 2 个字节，如图 4-4(b)所示。单引号与双引号是不同的概念，单引号只能括起单个字符，表示字符常量，而字符串常量必须使用双引号。



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| … | C | h | i | n | a | \0 | … |

(a) “China”

| | | | |
|---|---|---|---|
| … | A | \0 | … |

(b) “A”

图 4-4　字符串的存储

可以将一个字符串常量赋值给一个 char 型指针变量，其含义是将该字符串常量在内存中的首地址赋值给指针变量。例如，

```
char *p ;
p = "China" ;                    // 将字符串常量"China"赋值给指针变量 p
```

该赋值语句的含义是将字符串常量“China”在内存中的首地址赋值给指针变量 p，或者说是将指针变量 p 指向字符串常量“China”在内存中的首地址。

字符串常量可以包含转义字符。例如，

```
cout << "Yes\nNo" ;
```

字符串常量"Yes\nNo"包含一个转义字符'\n'，其含义是换行。上述 cout 语句在显示完"Yes"后将换一行再显示"No"。

如果字符串常量中有单引号、双引号或反斜杠，也需要用转义字符的形式。例如，

```
cout << "\"Yes\", \'No\'" ;          // 显示结果为："Yes", 'No'
```

因为 C++语言中的单引号、双引号和反斜杠被赋予了特殊含义，使用反斜杠\"、\'和\\是将它们还原成原来的含义，即字符串中确实包含了这样的符号。

## 4.4.2　字符数组

可以定义字符数组来保存字符串。例如，定义字符数组 str：

```
char str[10];  // 该数组有 10 个元素，最多可保存 10 个字符
```

### 1. 字符数组的初始化

定义字符数组时可以使用字符常量初始化数组元素，例如：

```
char str[10] = { 'C', 'h', 'i', 'n', 'a' } ;          // 初始化前 5 个元素，剩余元素自动初始化为'\0'
char str[ ] = { 'C', 'h', 'i', 'n', 'a' } ;           // 缺省下标时，根据初始值数量自动设为 5
```

定义字符数组时也可以使用字符串常量初始化数组元素，例如：

```
char str[10] = "China" ;                  // 初始化前 5 个元素，剩余元素自动初始化为'\0'
char str[ ] = "China" ;                   // 缺省下标时，根据字符串长度（含结束符）自动设为 6
                                          // 这 6 个元素分别被初始化为：'C'、'h'、'i'、'n'、'a'、'\0'
char str[3][10] = { "Tom", "John", "Mary" } ;   // 初始化二维字符数组（3 行 10 列）
```

2. 字符数组的赋值

字符数组在定义以后不能用字符串直接赋值，例如：

```
char str[10];
str = "China";                            // 语法错误
```

如需赋值，应改用如下的形式：

```
str[0] = 'C';   str[1] = 'h';   str[2] = 'i';   str[3] = 'n';   str[4] = 'a';
str[5] = '\0';                            // 增加结束符'\0'
```

3. 字符数组的整体输入和输出

通常，cin/cout 只能输入/输出单个数组元素，因此数组整体输入/输出需要使用循环结构逐个输入/输出各数组元素。但在输入/输出字符型数组时，C++语言对 cin/cout 指令做了特殊处理，使得 cin/cout 指令对字符型数组可以直接整体输入/输出，从而简化了字符数组的输入/输出编程。例如：

```
char str[10] ;          // 定义一个字符数组 str
cin >> str ;            // 整体输入字符数组 str
                        // 从键盘输入的字符个数应小于数组元素的个数，否则会导致越界错误
cout << str << endl ;   // 整体输出字符数组 str
```

4. 指针变量的输出

如果指针变量保存了另外某个变量的地址，可以使用 cout 显示该地址值，例如：

```
int x, *p = &x ;
cout << p << endl ;     // 显示指针变量 p 中保存的地址，即变量 x 的内存地址
```

如果指针变量是字符型，使用 cout 指令将显示其所指向的字符串，而不是其地址值。例如：

```
char str[10] = "China" ;        // 初始化前 5 个元素，剩余元素自动初始化为'\0'
char *p = str ;                 // 指针变量 p 指向字符数组 str 的首地址
cout << p << endl ;             // 执行该语句将从 str 的第 0 个元素开始逐个显示字符串的内容
                                // 直到结束符（即空字符）为止。显示结果：China
cout << p+2 << endl ;           // 执行该语句将从 str 的第 2 个元素开始显示，显示结果：ina
```

可以看出，cout 指令对字符型指针变量做了特殊处理。如果想显示字符型指针变量中保存的地址值，则需要将字符型指针强制转换成其他类型的指针，例如：

```
cout << (int *)p << endl;              // 显示字符型指针变量 p 中保存的地址值
```

其中，“int *”表示 int 型指针，也可以用“double *”“void *”等其他非 char 类型，其效果是一样的。

### 4.4.3　常用文字处理算法

#### 1. 求字符数组中字符串的长度

在字符数组中保存字符串，字符串是数组中的有效字符。字符串的长度不能超过数组的长度，否则会导致越界错误。所保存的字符串应当以空字符'\0'作为结束符，通过查找该结束符可以求出字符串的长度，如例 4-12 所示。

例 4-12　求字符数组中字符串长度的 C++程序

```
#include <iostream>
using namespace std;

int main( )
{
    char str[10] = "China" ;        // 字符数组 str 的长度为 10
                                    // 通过初始化保存字符串"China"，保存时以'\0'结束

    int n = 0;                      // 定义 int 型变量 n 来保存元素下标，初始化为 0
    while ( str[n] != '\0' )        // 通过循环结构从第 0 个元素开始逐个查找是否结束符
        n++ ;                       // 如果不是结束符，则下标加 1，继续查找下一个字符

    // 如果 str[n]是结束符则停止循环，此时 n 恰好就是字符串的长度（不含结束符）
    cout << n << endl;              // 显示字符串长度，显示结果：5
    return 0;
}
```

#### 2. 在字符串中插入一个字符

插入字符是文字处理的一个常用操作。假设当前字符数组 str 中存放的字符串是"Chna"，漏了一个字符 'i'。需要将 'i' 插入在第 2 个元素，即字符 'n' 的位置。插入操作需要将字符 'n'、'a' 依次后移，例 4-13 给出具体插入字符'i'的 C++程序。

例 4-13　在字符串中插入字符的 C++程序

```
#include <iostream>
using namespace std;

int main( )
{
    char str[10] = "Chna" ;         // 字符数组 str 的长度为 10
```

```
7                                // 通过初始化保存字符串"Chna"，漏了一个字符'i'
8
9       char ch = 'i' ;          // 定义变量 ch 保存需被插入的字符'i'
10      int n = 2;               // 定义变量 n 来保存元素下标，在第 2 个元素处插入'i'
11      char oldch ;             // 为什么要定义变量 oldch 呢？请看第 14 行注释
12      do                       // 通过循环结构从第 2 个元素开始插入 ch
13      {
14          oldch = str[n] ;     // 插入前要把当前位置原来的字符先暂存起来，以便后移
15          str[n] = ch ;        // 将 ch 插入到当前位置
16          ch = oldch ;    // 将 oldch 转存到 ch 中，算法含义是：将当前字符作为下一次循环时
17                    // 插入下一元素位置的字符，这样后移操作被转为在下一位置的插入操作
18          n++ ;                // 转入下一元素位置，循环做插入操作
19      }   while ( ch != '\0' ) ;   // 如果 ch 是结束符，则停止循环
20      // 循环结束后，从插入位置开始的字符都被后移了一位
21      str[n] = '\0' ;          // 为插入操作后的字符串添加结束符'\0'
22      cout << str << endl;     // 显示插入操作后的字符串，显示结果为：China
23      return 0;
24  }
```

### 3. 字符串复制

字符串复制（或称拷贝）就是将一个字符数组中的字符串复制到另一个字符数组中，例 4-14 给出一个字符串复制的 C++演示程序。

**例 4-14　字符串复制的 C++演示程序**

```
#include <iostream>
using namespace std;

int main( )
{
    char str1[10] = "China" ;     // 字符数组 str1 通过初始化保存字符串"China"
    char str2[20] ;          // 注意：定义 str2 时的数组长度应不小于字符串长度，否则会越界

    int n = 0;                    // 定义 int 型变量 n 来保存元素下标，初始化为 0
    while ( str1[n] != '\0' )     // 通过循环结构从第 0 个元素开始复制，遇到结束符停止
    {
        str2[n] = str1[n];        // 复制第 n 个元素
        n++ ;                     // 下标加 1，继续循环复制下一个元素
    }
    // 如果 str1[n]是结束符，则停止循环
    str2[n] = '\0' ;              // 为 str2 中的字符串添加结束符'\0'
    cout << str2 << endl;         // 显示复制到 str2 中的字符串
    return 0;
}
```

### 4. 字符串连接

字符串连接就是将分别保存在两个字符数组中的字符串连接起来，生成一个新的字符串，例 4-15 给出一个字符串连接的 C++演示程序。

**例 4-15　字符串连接的 C++演示程序**

```
#include <iostream>
using namespace std;

int main( )
{
    char str1[20] = "Hello " ;      // 字符数组 str1 通过初始化保存字符串"Hello "
    char str2[10] = "World!";       // 字符数组 str2 通过初始化保存字符串"World!"
    // 将 str2 中的字符串连接到 str1 中字符串的后面，生成一个新字符串"Hello World!"
    // 注意：定义 str1 时的数组长度要大于连接后新字符串的长度，否则会越界

    int n1 = 0;                     // 定义 int 型变量 n1 来保存 str1 的元素下标，初始化为 0
    while ( str1[n1] != '\0' )      // 通过循环结构查找 str1 中字符串的结束位置
        n1++ ;                      // 如果不是结束符，则下标加 1，继续查找下一个字符
    // 如果 str1[n1]是结束符，则停止循环，此时 n1 就是连接第 2 个字符串的连接点

    int n2 = 0;                     // 定义 int 型变量 n2 来保存 str2 的元素下标，初始化为 0
    while ( str2[n2] != '\0' )      // 通过循环结构从第 0 个元素开始将 str2 中的字符串
                                    // 连接到 str1 中字符串的后面（即 n1 所表示的连接点）
    {
        str1[n1+n2] = str2[n2];     // 将 str2 中的第 n2 个元素复制到 str1 中的第 n1+n2 个元素
        n2++ ;                      // 下标 n2 加 1，继续复制下一个元素
    }
    // 如果 str2[n2]是结束符，则停止循环
    str1[n1+n2] = '\0' ;            // 为 str1 中的新字符串添加结束符'\0'
    cout << str1 << endl;           // 显示连接后 str1 中的新字符串，显示结果：Hello World!
    return 0;
}
```

## 本节习题

1. 存储下列常量，占用字节数最少的是（　　）。

   A. 8　　B. 8.0　　C. '8'　　D. "8"

2. 若有字符数组定义“char a[ ] = "China";”，则数组 a 有几个数组元素？（　　）

   A. 0 个　　B. 5 个　　C. 6 个　　D. 不确定

3. 若有字符数组定义“char a[ ] = "中国";”，则数组 a 有几个数组元素？（　　）

   A. 2 个　　B. 3 个　　C. 4 个　　D. 5 个

4. 下列字符数组定义语句中，语法错误的是（　　）。

   A. char a[3];　　B. char a[ ] = "ABC";

   C. char a[3] = { 'A', 'B', 'C' };　　D. char a[3] = "ABC";

5. 执行 C++语句“char a[ ] = "Hello\0World"; cout << a;”，显示器将显示（ ）。
   A. Hello\0World B. Hello World C. Hello D. World
6. 执行 C++语句“char a[ ] = "HelloWorld"; cout << a+5;”显示器将显示（ ）。
   A. HelloWorld B. HelloWorld5 C. Hello D. World

# 4.5 中文处理

计算机系统处理中文需要满足以下 3 个方面的条件。

（1）制定**汉字字符编码标准**。例如《信息交换用汉字编码字符集》，即国标 GB 2312。

（2）支持上述编码标准的**中文操作系统**。它能提供汉字输入法以及显示或打印用的汉字字形库，例如中文版 Windows 操作系统。

（3）支持中文处理的**应用软件**。用户使用中文版的应用软件进行中文处理，例如中文版 Word 文字处理软件。

其中，前两项是中文处理的基础。有了这两个基础，程序员才可以编写第(3)项用户所使用的中文版应用软件。可以在 C++程序中直接书写中文字符串常量，如例 4-16 所示。

**例 4-16 含中文字符串常量的 C++演示程序**

```
#include <iostream>
using namespace std;

int main( )
{
    char str1[ ] = "China" ;            // 字符数组 str1 保存英文字符串"China"
    char str2[ ] = "中国";              // 字符数组 str2 保存中文字符串"中国"
    char str3[ ] = "中国, China";       // 字符数组 str3 保存中英文混合字符串"中国, China"

    cout << str1 << endl;              // 显示保存在 str1 中的英文字符串
    cout << str2 << endl;              // 显示保存在 str2 中的中文字符串
    cout << str3 << endl;              // 显示保存在 str3 中的中英文混合字符串
    return 0;
}
```

从例 4-16 中可以看出，中文字符串和英文字符串在使用上没有什么区别。但由于汉字字符的编码标准与英文字符不同，汉字字符在存储和处理算法上将会有所不同。为了编写中文处理软件，程序员首先需要深入理解汉字字符的编码方法。

## 4.5.1 字符编码标准

ASCII 码的字符集包含以英文字母为主的 128 个字符与符号，只能用于英文处理。为了处理本国语言，世界各国分别制定了本国或本地区的文字编码标准，例如中国制定了 GB 2312 汉字编码标准。汉字编码首先要确定有哪些汉字字符（即字符集），然后再确定每个字符的编码值。

ASCII 码使用单字节（8 位）编码，只有 256 个码值（0~255），最多只能为 256 个字符编码。汉字是象形文字，有几万个字符，常用的也有数千个，因此汉字编码需要使用双字节（16 位）。理论上双字节编码可以提供 $2^{16}$=65 536 个码值，能为 65 536 个字符编码。ASCII 码的字符集被称为单字节字符集（Single-Byte Character Set，SBCS），汉字编码的字符集被称为双字节字符集（Double-Byte Character Set，DBCS）。日文、韩文的处理方法和中文类似，国际上通常将这 3 种文字统称为 CJK，即中文（Chinese）、日文（Japanese）和韩文（Korean）的首字母缩写。

操作系统提供了文字处理相关的输入法、编码、显示和打印字库等基础功能。通常，操作系统可同时处理英文和一种非英语文字，例如中文版 Windows 可同时处理英文和中文，日文版 Windows 可同时处理英文和日文。英文符号使用单字节 ASCII 编码标准，非英文符号则分别使用各国自己制定的双字节编码标准，这种混合编码方法被称为 **ANSI 编码**，或 **MBCS（Multi-Byte Character Set）编码**。

基于 ANSI 编码的计算机系统存在一个缺陷，即只有英文可以与其他语种混用，例如英文可以与中文或日文混用，但中文与日文不能混用。因此开发多语种软件时需要分别开发多个不同语种的版本，例如中文版、日文版等。为了解决这个问题，相关国际组织制定了一种新的统一的编码标准，称为 **Unicode 编码**标准，或 **ISO 10646 标准**。Unicode 编码将世界上主要的语言文字合在一起，构建一个大的字符集，然后统一进行编码。未来，开发软件产品应当基于 Unicode 编码进行开发，这样可以更方便地推广到全球市场。

## 4.5.2　基于 ANSI 编码的中文处理程序

一个计算机程序是否支持中文处理有两层含义。

（1）是否具有适合中国用户使用的中文界面。

（2）是否能够处理中文，例如对中文字符串的插入、删除等操作。

为了编写能够处理中文的 C++程序，程序员首先需要了解汉字编码标准。

### 1. 基于 ANSI 编码的汉字编码标准

为了处理中文，首先要建立中文文字的编码标准。中文编码标准首先要确定有哪些字符（即字符集），然后再确定每个字符的编码值。1980 年，中国国家标准总局发布《信息交换用汉字编码字符集》，即 **GB 2312 标准**。该标准共收入 6763 个常用汉字和 682 个图形字符。

1995 年，全国信息技术标准化技术委员会对 GB 2312 标准进行了扩充，制定出《汉字内码扩展规范》，即目前常用的 **GBK 标准**（K 是“扩展”汉语拼音的第一个字母）。该标准为 GBK 1.0 版本，共收录 21 886 个汉字（含部分繁体和一些图形符号）。2000 年又修订推出了 GBK 18030 标准，共收录 27 484 个汉字，同时还收录了藏文、蒙文、维吾尔文等少数民族文字。现在的中文 Windows 操作系统都支持 GBK 18030 编码，某些嵌入式系统可能还在使用 GB 2312 标准。

基于 ANSI 编码标准的中文 Windows 操作系统包含两套字符集，一是 ASCII 码字符集（半角字符），使用单字节存储，码值范围为 0~127（十六进制为 0x00~0x7F），存储时字

节的最高位都为 0；二是 GBK 编码字符集（全角字符），使用双字节存储，其中第 1 个字节称为前导字节，存储时该字节的最高位都为 1。C++程序可以根据字节最高位来判断字符类型，0 表示 ASCII 码字符（即英文字符），1 表示 GBK 字符（即汉字字符）。图 4-5 给出字符串“中国 abc”的 ANSI 编码存储示意图，其中每个英文字符占 1 个字节，而中文字符占 2 个字节。字符串的结束标记仍为 0x00，即空字符。

| | |
|---|---|
| 中 | 0xD6 |
| | 0xD0 |
| 国 | 0xB9 |
| | 0xFA |
| a | 0x61 |
| b | 0x62 |
| c | 0x63 |
| | 0x00 |

图 4-5　字符串"中国 abc"的 ANSI 编码存储示意图

### 2. 编写具有中文界面的 C++程序

在 C++语言中，中文字符串和英文字符串在以下几个方面是完全一样的：

- 都使用字符数组来存储。
- 都使用空字符'\0'作为字符串结束标记。
- 都可以使用 cin 指令直接从键盘输入，并保存到某个字符数组中。
- 都可以使用 cout 指令将保存在某个字符数组中的字符串输出到显示器上。

C++语言支持用双引号括起来的中文字符串常量，但不支持用单引号括起来的单个中文字符常量。例如“中”是正确的，而'中'是错误的。在 C++语言看来，一个中文字符占 2 个字节，相当于 2 个英文字符，应当是字符串。例 4-17 给出一个具有中文界面的 C++温度转换程序。

**例 4-17　具有中文界面的 C++温度转换程序**

```
#include <iostream>
using namespace std;

int main( )
{
    double ctemp, ftemp;
    cout << "请输入摄氏温度：";          // 在需要用户输入摄氏温度前显示中文提示信息
    cin >> ctemp;                       // 从键盘输入摄氏温度
    ftemp = ctemp * 1.8 + 32;
    cout << "华氏温度等于 " << ftemp << endl;  // 显示换算结果
    return 0;
}
```

### 3. 编写能处理中文的 C++程序

C++语言使用字符数组保存字符串，可以是英文、中文，也可以是中英文混合的字符串。按照 ANSI 编码的原理，可以根据字节最高位来判断字符类型，0 表示 ASCII 码字符（即英文字符），1 表示 GBK 字符（即中文字符）。例 4-18 给出一个筛选中文字符的 C++演示程序。

**例 4-18　筛选中文字符的 C++演示程序**

```
#include <iostream>
using namespace std;

int main( )
{
    char str[20];                  // 定义一个字符数组 str，保存从键盘输入的字符串
    cin >> str;                    // 从键盘输入一个中英文混合的字符串，例如输入"中国 abc"

    char cstr[20];     // 再定义一个字符数组 cstr，保存从 str 中筛选出的中文字符
    int n, cn;         // 预先定义好循环要用到的变量，n 表示 str 的下标，cn 表示 cstr 的下标
    n = 0;   cn = 0;               // 循环开始前，n 和 cn 的初始值设为 0
    while (str[n] != '\0')         // 空字符'\0'是字符串的结束标记
    {
        if ( (str[n] & 0x80) != 0 )         // 通过位运算检查最高位是否为 0
        {
            cstr[cn] = str[n];   cstr[cn+1] = str[n+1];      // 最高位为 1：中文字符，复制
            cn += 2;   n += 2;              // 中文字符：下标加 2，转到下一个字符
        }
        else   // 最高位为 0：英文字符，不复制
            n++;                            // 英文字符：下标加 1，转到下一个字符
    }
    cstr[cn] = '\0';                          // 为 cstr 添加字符串结束标记
    cout << cstr << endl;    // 显示筛选出的中文字符，"中国 abc"的筛选结果为"中国"
    return 0;
}
```

## 4.5.3　基于 Unicode 编码的中文处理程序

Unicode 编码将世界上主要的语言文字合在一起，构建一个大的字符集，然后统一进行编码。Unicode 字符集也称为**通用字符集**（Universal Character Set，简称 UCS），目前已收录超过十万个字符，其中包括原 ASCII 码的字符集，这些字符的码值保持不变，码值范围仍为 0~127。Unicode 字符集也收录了 GBK 中文字符集，但每个字符的码值不一样了，码值范围是 0x4E00 ~ 0x9FBF（十六进制），例如汉字“中”的 GBK 编码为 0xD6D0，而其 Unicode 编码为 0x4E2D。Unicode 字符集还收录了日文、韩文、阿拉伯文等数十种语言文字。

存储一个 Unicode 字符需要占用几个字节？不同操作系统或不同编程语言可能是不一样的。例如 VC 6.0 使用 2 个字节（16 位）来存储一个 Unicode 字符，我们就说 VC 6.0 实

现 Unicode 编码的方式是 16 位。Unicode 编码主要有 3 种实现方式，分别是 UTF-8 格式、UTF-16 格式和 UTF-32 格式，其中 UTF 表示 Unicode 转换格式（Unicode Transformation Format），8、16 和 32 表示存储位数，即 8 位、16 位和 32 位。

UTF-32 格式使用 4 个字节存储 Unicode 编码。UTF-32 格式是定长编码，简单、直接，但占用存储空间多，例如存储一个英文字符也需要 4 个字节。

UTF-8 格式将 Unicode 编码划分成 4 个区间，分别进行再编码，最后形成一种变长编码，其长度为 1~4 个字节不等。使用 UTF8 编码存储一个英文字符只需 1 个字节，而汉字字符则需要 3~4 个字节。UTF8 是变长编码，会增加文字处理算法（例如插入、删除操作）的复杂性。

UTF-16 格式介于 UTF-8 和 UTF-32 之间，存储一个字符（无论中英文）都是 2 个字节，它也是一种定长编码。基于定长编码的文字处理算法比较简单。目前 VC 6.0、Java 等计算机语言大多使用 UTF-16 格式来存储 Unicode 编码。

### 1. 宽字符类型 wchar_t

C++语言将 Unicode 编码的字符称为**宽字符**，并专门定义了一种新的宽字符类型，关键字为 wchar_t。VC 6.0 中的 wchar_t 类型占用 2 个字节，按 unsigned short 格式存储 UTF-16 格式的 Unicode 编码。

C++语言使用字母“L”来指定宽字符常量，例如 L'a'表示宽字符常量'a'，L'中'表示宽字符常量'中'。单个中文字符不能是字符常量，但可以是宽字符常量。C++语言同样使用字母“L”来指定宽字符串常量，例如 L"中国 abc"表示一个宽字符串常量。

编译 C++源程序时，VC 6.0 编译器将程序中的宽字符常量转换成 UTF-16 格式的 Unicode 编码。图 4-6 给出宽字符串 L"中国 abc"的 Unicode 编码存储示意图，其中每个英文字符和中文字符都占用 2 个字节。宽字符串的结束标记为 0x0000，即宽空字符。

| | |
|---|---|
| 中 | 0x4E2D |
| 国 | 0x56FD |
| a | 0x0061 |
| b | 0x0062 |
| c | 0x0063 |
| | 0x0000 |

图 4-6　宽字符串 L"中国 abc"的 Unicode 编码存储示意图

可以定义宽字符类型的变量或数组，每个宽字符变量或数组元素占 2 个字节。例如：

```
wchar_t wc = L'中';   // 定义一个宽字符变量 wc，初始化为'中'
wchar_t wstr[ ] = L"中国 abc";   // 定义一个宽字符数组变量 wstr，初始化为"中国 abc"
                                 // 数组 wstr 的元素个数被设置成 6，其中包含 1 个宽空字符
```

### 2. 基于 UTF-16 格式的 C++中文处理程序

使用 UTF-16 格式存储一个英文字符或一个中文字符都是 2 个字节，是定长编码。基于定长编码的文字处理算法比较简单，例 4-19 给出一个删除字符串中某个字符的 C++演示程序。

**例 4-19 删除字符串中某个字符的 C++演示程序**

```
1   #include <iostream>
2   #include <locale>
3   using namespace std;
4   
5   int main( )
6   {
7       wchar_t wstr[20] = L"中国 abc";     // 定义宽字符数组 wstr，初始化为"中国 abc"
8   
9       int n;                             // 预先定义好循环要用到的变量 n，表示 wstr 的下标
10      n = 2;                             // 删除数组 wstr 中的字符'a'，其下标为 2
11                                         // 删除某个字符，就是将其后的字符依次往前移一位
12      while ( wstr[n] != L'\0' )         // 宽空字符 L'\0'是宽字符串的结束标记
13      {
14          wstr[n] = wstr[n+1];           // 将其后面的字符往前移一位
15          n++;                           // 下标加 1，转到下一个字符
16      }
17  
18      wcout.imbue( locale("chs") );      // 将语言设置为简体中文 chs
19      wcout << wstr << endl;             // 显示删除'a'后的字符串，显示结果为"中国 bc"
20                                         // 显示宽字符串时改用 wcout 指令
21      return 0;
22  }
```

输出宽字符串需改用 wcout 指令。wcout 指令在显示中文时，首先将 Unicode 编码转换成 GBK 编码，然后再显示 GBK 编码的中文字符。使用 wcout 指令之前需将语言设置为简体中文（即 GBK 编码），如例 4-19 中代码第 18 行所示。

输入宽字符串需改用 wcin 指令。wcin 指令在输入中文时，先转换成 GBK 编码，然后再将 GBK 编码转换成 Unicode 编码。使用 wcin 指令之前需将语言设置为简体中文（即 GBK 编码），例如：

```
wchar_t wstr[20];
wcin.imbue( locale("chs") );           // 将语言设置为简体中文 chs
wcin >> wstr;
```

**注：** VC 6.0 的 wcin 指令有 bug（错误），在微软后续推出的 Microsoft Visual Studio 集成开发环境中才能正常使用 wcin 指令。

### 本节习题

1. 常用的中文编码标准不包括下列哪种编码？（　　）

   A. GB 2312　　B. GBK　　C. ASCII　　D. Unicode

2. 在 C++语言中，使用 ANSI 编码存储字符串“农大 CAU”需要几个字节？（　　）

   A. 5　　B. 6　　C. 7　　D. 8

3. 在 C++语言中，使用 UTF-16 编码存储字符串“农大 CAU”需要几个字节？（　　）

   A. 5　　B. 8　　C. 10　　D. 12

4. 字符串常量“农业 CAU”的宽字符书写形式是（　　）。

   A. "农大 CAU"　　B. L"农大 CAU"

   C. w"农大 CAU"　　D. W"农大 CAU"

5. 在 VC 6.0 中，wchar_t 类型的存储位数与下列哪种数据类型相同？（　　）

   A. char　　B. unsigned short　　C. long　　D. double

## 4.6 程序设计方法简介

这里我们对前 4 章的学习内容进行一个简单回顾。

第 1 章我们学习了计算机硬件结构、计算机程序及其开发过程；还学习了计数制、数据存储及数据类型，并大致了解了什么是 C++语言。

第 2 章以数值计算问题为例，学习了程序中的变量和常量、算术运算、位运算和赋值运算；学习了如何使用 cin、cout 指令来输入/输出数据；还学习了访问变量内存单元的 3 种方式，即变量名、引用和指针。

第 3 章学习了算法与控制结构，重点学习了如何使用 C++语言中的选择语句和循环语句来分别描述选择结构和循环结构算法；还学习了如何通过布尔类型、关系运算和逻辑运算来描述算法中的条件；最后通过几个程序实例学习了算法的设计方法和评价标准。

第 4 章我们学会了如何使用数组来保存数据集合，并初步学习了常用的数组处理算法；还具体分析了指针与数组的关系，在此过程中进一步加深了对指针的理解；最后学习了如何通过字符类型、字符数组来进行文字处理。

经过前 4 章的学习，读者已掌握了程序设计原理基础部分的内容，并可以使用 C++语言编写简单的数值计算程序和文字处理程序。实际上，还有很多数据处理问题都可以归为数值计算问题。例如，图像处理就是一个数值计算问题。数字化后的图像数据就是一个矩阵，可以定义二维数组来保存数字图像，每个数组元素对应一个像素。图像处理就是对二维数组中的数组元素进行数值计算，例如修改数组元素的值就是调整图像的亮度或色彩。程序员可以编写 C++程序，通过二重循环遍历处理二维数组就可以实现图像处理的功能。

那么该如何编写更大型的计算机程序呢？这就需要进一步学习程序设计原理的高级部分，即程序设计方法。程序的功能是数据处理，其中包括数据和算法两大部分。数据是程序处理的对象，对应程序中的变量或常量。算法是描述数据处理过程的一组操作步骤，这

就是程序中所编写的一组语句序列。大型程序的功能很强，这意味着要处理大量的数据，数据处理的算法也很多、很复杂。程序设计方法的**基本思想**是：将大型程序中的数据和算法分解成程序零件，将不同零件的设计任务交由不同的程序员完成，这样就能以团队的形式来共同开发，然后将开发好的零件组装在一起，最终完成复杂的程序功能。目前，程序设计方法分为结构化程序设计和面向对象程序设计两种，分别采用不同的方式来分解和组装程序零件。

更进一步地，如果所分解出的程序零件在以前项目中曾经开发过，或者可以从市场上购买到，那么就可以直接使用这些零件来组装软件，实现快速开发。使用已有的程序零件，实际上是重用其程序代码，这就是程序设计中的**代码重用**（code reuse）。为了让不同程序员开发的程序零件能够正确地组装在一起，在编写时它们应遵守共同的语法规则。因为易于复制，所以代码重用的成本很低，这是软件行业所独有的特点。代码重用可以极大地提高软件开发效率，代码重用也因此成为软件技术不断进步的主要动力。

为了应用程序设计方法来编写大型复杂程序，计算机语言需要提供描述和组装程序零件的语法规则。支持结构化程序设计方法的语言称为结构化程序设计语言，支持面向对象程序设计方法的语言称为面向对象程序设计语言。C 语言是一种结构化程序设计语言，Java 语言是一种面向对象程序设计语言。而 C++语言既支持结构化程序设计方法，又支持面向对象程序设计方法。本书下面的章节将分别介绍结构化程序设计方法和面向对象程序设计方法，并具体介绍 C++语言中相关的语法规则。

**学习本章的要点**

- 读者要重点掌握数组定义及访问的语法规则。
- 读者要认识到计算机内部对数组的管理和访问是通过指针（即内存地址）来实现的。
- 读者应通过案例学习初步掌握常用的数组处理算法。

## 4.7　本章习题

1. **阅读程序**。阅读下列 C++程序。阅读后请说明程序的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
using namespace std;
int main( )
{
    int a[10] = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };
    int N = 9;
    int n, t;
    for (n = 0; n <= N; n++)
    {
        if (n >= N–n)
            break;
        t = a[n];    a[n] = a[N–n];    a[N–n] = t;
    }
```

```
    for (n = 0; n <= N; n++)
        cout << a[n] << ", ";
    cout << endl;
    return 0;
}
```

2. **程序改错**。阅读下列 C++程序，并检查其中的语法错误。修改错误，并保证程序的功能不变。

```
#include <iostream>
using namespace std;
int main( )
{
    char str[ ] = '1, 2, 3, good morning!';     // 定义一个字符串数组 str，并初始化
    int n = 0;
    while (str[n] != '0')
    {
        if (str[n] >= a && str[n] <= z)     // 如果是小写英文字母，则改成大写
            str[n] −= 32;
        n− −;                               // 继续检查下一个字符
    }
    cout << *str << endl;                   // 显示转换后的字符串
    return 0;
}
```

3. **编写程序**。编写一个 C++程序，生成如下等差数列的前 10 项：$a_0 = 1$，$a_n - a_{n-1} = 3$，并保存到数组中。显示该数列的生成结果，以及前 5 项之和。

4. **编写程序**。编写一个 C++程序，当用户输入一个英文字母后，程序能够按照字母表的顺序显示出 3 个相邻的字母，其中用户输入的字母在中间。例如用户输入字母 d，则程序输出 cde；如果用户输入字母 Z，则程序输出 YZA（即字母 A 和 Z 被认为是相邻的）。

5. **编写程序**。恺撒加密法的加密规则是：将原来小写的字母用字母表中其后第 3 个字母的大写形式来替换，大写字母按同样规则用小写字母替换。字母表可看成是首末衔接的。例如字母 c 就用 F 来替换，字母 y 用 B 来替换，而字母 Z 则用 c 代替。编写一个 C++程序实现输入一个字符串，输出将其加密后的密码。运行示例：

```
AMDxyzXYZ<回车键>
dpgABCabc
```

# 第5章 结构化程序设计之一

程序描述了某种数据处理的过程和步骤。数据是程序处理的对象。一个复杂的程序设计任务可能要处理大量数据，为此C++语言提供了数组的语法形式。数组可以存储大量同类型的数据。将数据处理的过程细分成一组严格的操作步骤，这组操作步骤被称为算法。如果数据处理的算法很长、很复杂，该如何来编写这样很长、很复杂的程序呢？

**结构化程序设计方法**就是将一个复杂算法分解成多个简单的模块，分而治之，然后将这些模块组装起来，最终形成一个完整的数据处理流程。C++语言支持结构化程序设计方法，以函数的语法形式来描述和组装模块，即**函数的定义和调用**。

结构化程序设计在将一个数据处理过程分解成多个算法模块之后，各模块之间需要共享数据。C++语言提供了**分散管理**和**集中管理**这两种数据管理策略。

## 5.1 结构化程序设计方法

结构化程序设计方法也称为面向过程的程序设计方法。**结构化程序设计方法**的基本原理是：将一个求解复杂问题的过程划分为若干个**子过程**，每个子过程完成一个独立的、相对简单的功能；用算法描述各个过程的操作步骤，每个算法称为一个**模块**；采用“**自顶向下，逐步细化**”的方法逐步分解和设计算法模块，再通过**调用关系**将各个模块组装起来，最终形成一个完整的数据处理流程。采用结构化程序设计方法，程序员重点考虑的是如何分解和设计算法。

### 5.1.1 设计举例

**设计任务：**公园计划修建一个长方形观赏鱼池，另外配套修建一大一小两个圆形蓄水池，分别存放清水和污水（见图5-1）。养鱼池和蓄水池的造价均为10元/$m^2$。请设计一个测算养鱼池工程总造价的算法。

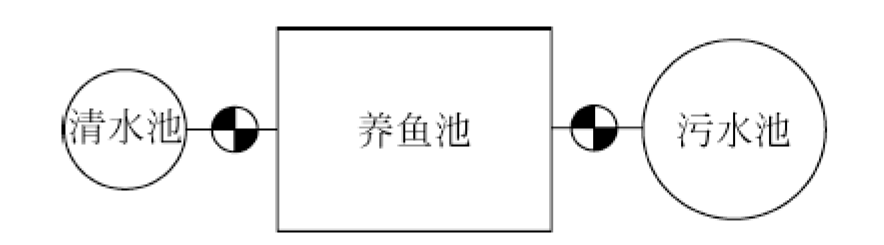



图5-1　公园观赏鱼池

结构化程序设计一般分为两个阶段，分别是概要设计和详细设计。设计结果通常以自然语言或流程图的形式来描述，并编写成书面的程序设计报告。

### 1. 概要设计

测算养鱼池工程总造价的算法内容主要包括：输入原始数据，分别测算养鱼池、清水池和污水池的造价，汇总并显示总造价。把上述算法内容设计成一个粗略的算法（称为概要设计），用自然语言描述出来，如例 5-1 所示。

| 例 5-1　测算养鱼池工程总造价的算法（概要设计） | |
| --- | --- |
| 1 | 定义变量，申请保存原始数据（包括养鱼池的长和宽、清水池和污水池的半径）和计算结果（工程总造价）所需的内存空间。 |
| 2 | 从键盘输入原始数据，包括养鱼池的长和宽、清水池和污水池的半径。 |
| 3 | 计算长方形养鱼池的造价，累加到工程总造价。 |
| 4 | 计算圆形清水池的造价，累加到工程总造价。 |
| 5 | 计算圆形污水池的造价，累加到工程总造价。 |
| 6 | 在显示器上显示测算出的工程总造价。 |

例 5-1 中第 3~5 步的计算长方形养鱼池造价、圆形清水池造价和圆形污水池造价这 3 个算法子过程并没有展开，要将它们提取出来进一步细化，形成独立的模块。测算圆形清水池和圆形污水池造价的算法完全一样，可以共用同一个算法模块，即计算圆形水池造价的模块。共用算法模块可以减少重复设计。这样例 5-1 的算法可以被分解成 3 个算法模块，分别是一个主模块、一个计算长方形养鱼池造价的子模块和一个计算圆形水池造价的子模块。

### 2. 详细设计

划分好模块之后还需进一步细化各模块的算法，细化到每个操作步骤基本上可以对应计算机语言中的某条指令为止，这被称为是详细设计。例 5-2 给出细化后的详细设计结果。

例 5-2 中，子模块 1 描述了测算长方形养鱼池造价的算法，该算法有两个输入参数（用于接收养鱼池的长和宽），有一个计算结果（即养鱼池的造价）；子模块 2 描述了测算圆形水池造价的算法，该算法有一个输入参数（用于接收水池的半径），有一个计算结果（即水池的造价）。主模块负责输入原始数据（包括养鱼池的长和宽、清水池和污水池的半径）、计算并显示最终的工程总造价。主模块在计算工程造价时分别调用了子模块 1（第 3 步）和子模块 2（第 4、第 5 步），其中子模块 2 被调用了两次。

主模块每次调用子模块时都有两次数据传递：第一次是主模块将原始数据作为输入参数传递给子模块，子模块接收输入参数；第二次是子模块将计算结果返回给主模块，主模块接收返回结果。例 5-2 所示的算法通过调用关系将 3 个模块组织起来，最终形成了一个测算养鱼池工程总造价的完整求解流程。

## 5.1.2　基于模块的团队分工协作开发

结构化程序设计将一个复杂算法分解成多个模块。可以将不同模块的设计任务交给不同程序员去完成，然后再通过调用关系将这些模块组织起来，最终形成一个完整的算法流程，这就是基于模块的**团队分工协作开发**模式。

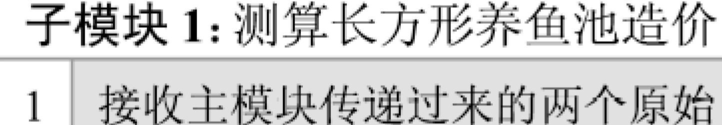
结构化程序设计采用“自顶向下，逐步细化”的方法逐步分解和设计算法模块。分解出的模块可能还比较复杂，可以进一步分解，即多级分解。随着模块的逐级向下分解，下级模块的功能越来越单一，也越来越通用。例如，例 5-2 子模块 1 中的测试长方形养鱼池造价算法可以再分解出一个专门求长方形面积的子模块 3，子模块 2 中的测试圆形水池造价算法也可以再分解出一个专门求圆面积的子模块 4。图 5-2 给出了再次分解后各模块之间的调用关系图。凡是需要求长方形面积的地方都可以调用子模块 3，需要求圆面积的地方都可以调用求子模块 4。这两个子模块的功能单一，但通用性非常强。

**例 5-2　测算养鱼池工程总造价的算法（详细设计）**

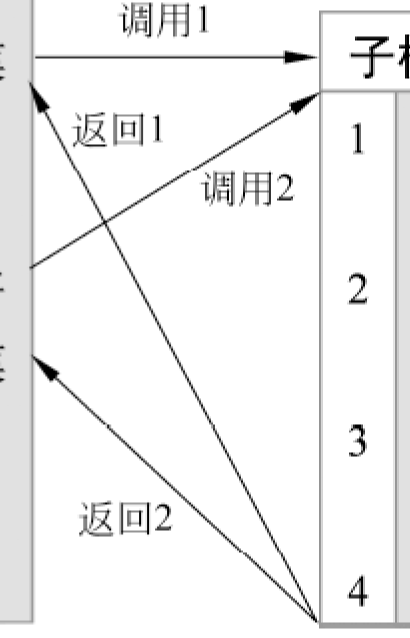
| **主模块**（测算养鱼池工程总造价） | |
|---|---|
| 1 | 定义变量，申请保存原始数据（包括养鱼池的长和宽、清水池和污水池的半径）和计算结果（工程总造价）所需的内存空间。 |
| 2 | 从键盘输入原始数据，包括养鱼池的长和宽、清水池和污水池的半径。 |
| 3 | **调用子模块** 1：跳转到子模块 1，跳转前将长方形养鱼池的长、宽数据传递过去，然后将子模块 1 返回的计算结果累加到工程总造价。 |
| 4 | **调用子模块** 2：跳转到子模块 2，跳转前将圆形清水池的半径传递过去，然后将子模块 2 返回的计算结果累加到工程总造价。 |
| 5 | **调用子模块** 2：跳转到子模块 2，跳转前将圆形污水池的半径传递过去，然后将子模块 2 返回的计算结果累加到工程总造价。 |
| 6 | 在显示器上显示测算出的工程总造价。 |

调用

返回



| **子模块 1**：测算长方形养鱼池造价 | |
|---|---|
| 1 | 接收主模块传递过来的两个原始数据，即长方形养鱼池的长和宽。 |
| 2 | 再定义一个变量，申请保存造价计算结果所需的内存空间。 |
| 3 | 按照公式计算造价：<br>造价=长×宽×单价。 |
| 4 | 将计算结果返回给主模块。 |




| **子模块 2**：测算圆形水池造价 | |
|---|---|
| 1 | 接收主模块传递过来的一个原始数据，即圆形水池的半径。 |
| 2 | 再定义一个变量，申请保存造价计算结果所需的内存空间。 |
| 3 | 按照公式计算造价：<br>造价=3.14×半径×半径×单价。 |
| 4 | 将计算结果返回给主模块。 |

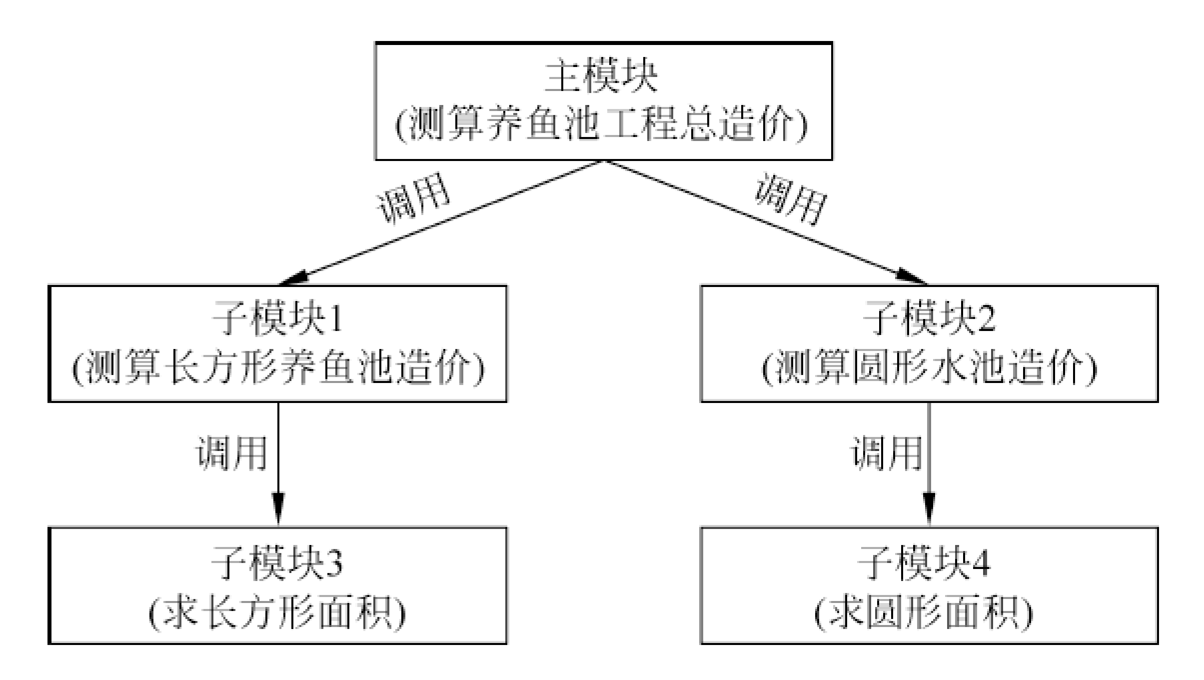



图 5-2 例 5-2 算法进一步分解后的模块调用关系图

每个上级模块可以分解出多个下级模块。不同上级模块可能会分解出功能完全相同的下级模块，即重复的模块。结构化程序设计应当合并重复模块，避免重复劳动。合并后的模块可以被不同的上级模块调用。同一模块可以被多个模块调用，或被同一模块调用多次，这称为模块的**重用**或**共用**。图 5-3 给出了结构化程序设计中常见的模块调用关系图。

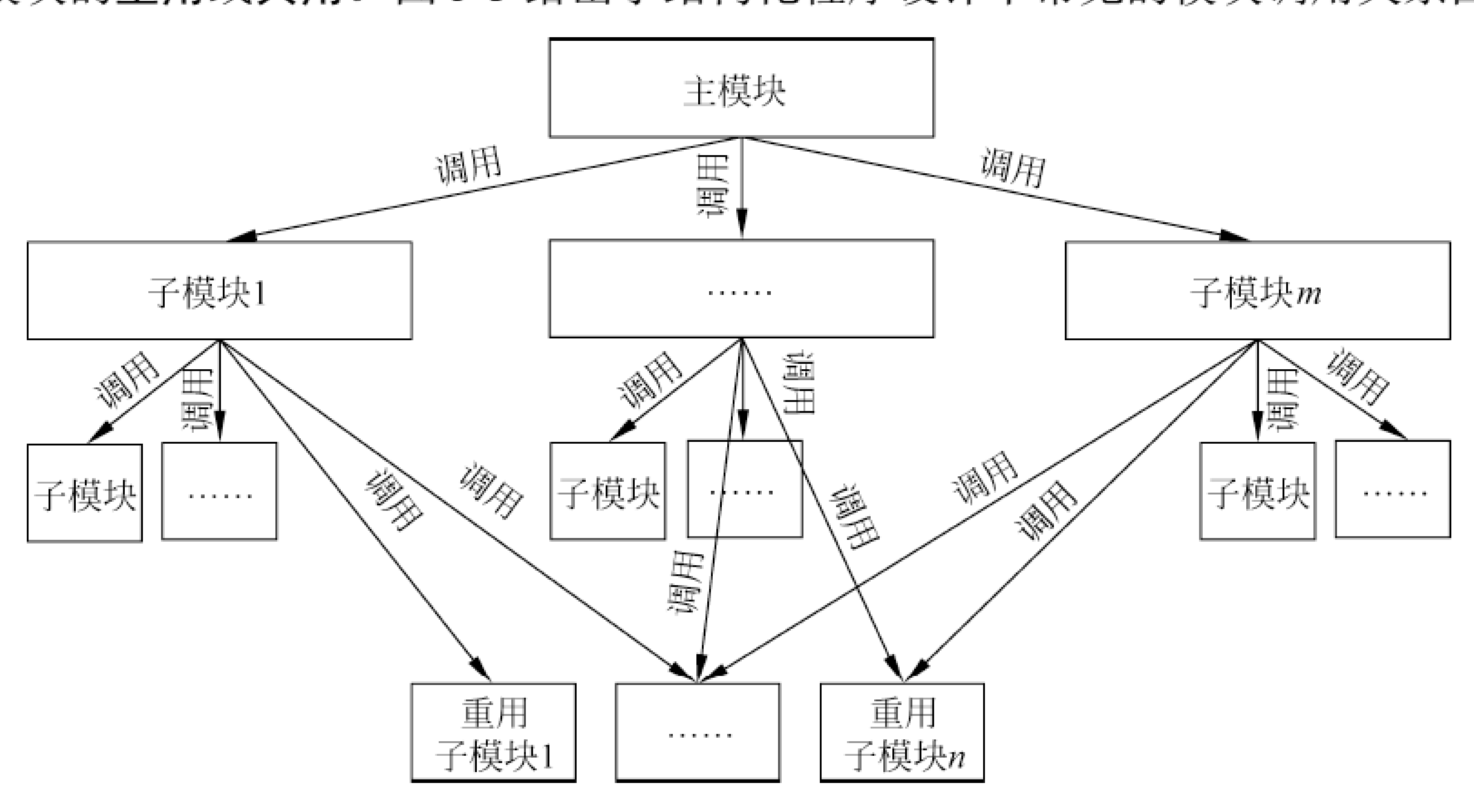



图 5-3 常见的模块调用关系图

调用其他模块的模块称为**主调模块**，被其他模块调用的模块称为**被调模块**。可以将不同模块的设计任务交给不同程序员去完成。假设程序员甲负责主调模块，程序员乙负责被调模块，如图 5-4 所示。

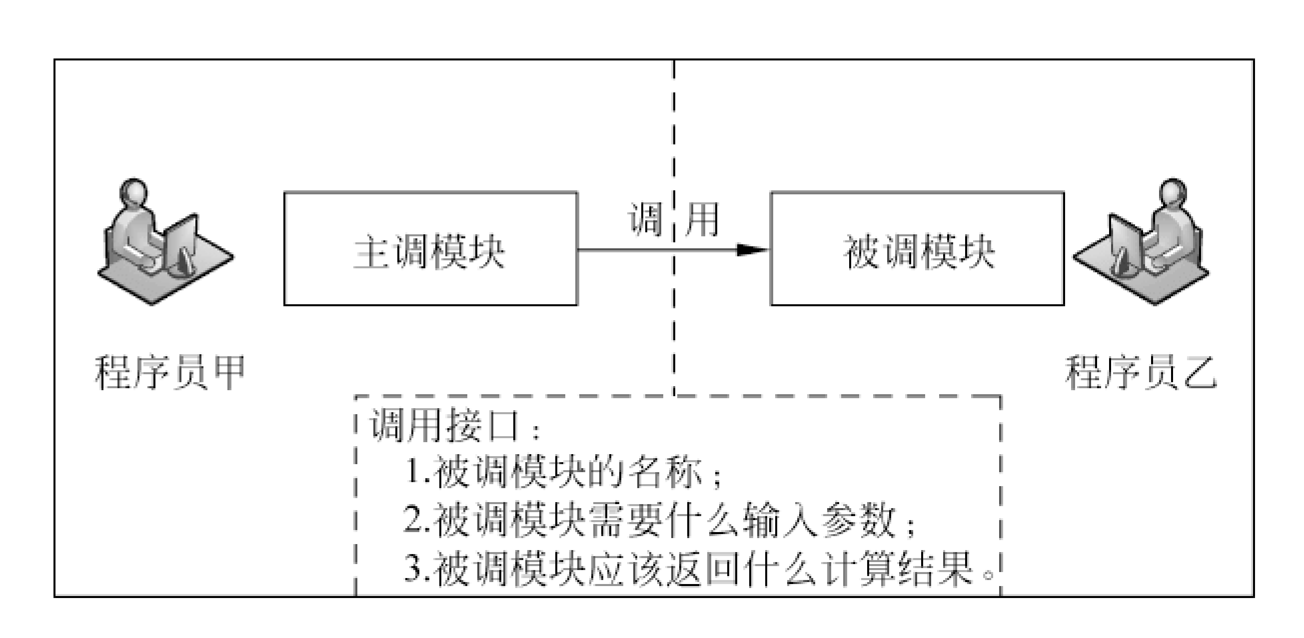



图 5-4 主调模块与被调模块

程序员甲在设计主调模块时需调用程序员乙的模块。甲、乙两位程序员应事先协商好被调模块的**调用接口**，其中包括被调模块的**名称**、需要什么**输入参数**、应该返回什么计算结果（称为**返回值**）共 3 项内容。程序员甲在调用时只关心被调模块的调用接口，不会关心乙是如何实现其内部算法的。程序员乙在设计被调模块时将专注于内部算法设计，并按照调用接口的规定指明模块名称，接收原始数据并返回正确的结果。程序员乙也不会关心甲什么时候、在什么地方调用自己的模块。

结构化程序设计为团队分工协作，开发大型软件提供了一种科学有效的方法，具体体现在以下三个方面。

（1）**模块化是团队分工的基础**。只有将程序设计任务分解成模块之后，才能对开发团队进行分工，将不同模块交给不同程序员去完成。在确定好模块之间的调用接口之后，各程序员独立开展工作，互不干扰，可并行开发。

（2）**模块接口是团队协作的基础**。设计被调模块的程序员应当按照调用接口的规定指明模块名称，接收原始数据并返回正确的结果。设计主调模块的程序员在调用时也应当按照调用接口的规定指明被调模块的名称，传递原始数据并接收返回结果。团队中的程序员都必须遵守调用接口的规定，这样所设计的模块才能被正确地组装在一起，协同工作。

（3）**模块重用可以极大地提高软件开发效率**。同一模块可以被多个模块调用，或被同一模块调用多次，这就是模块重用。程序主模块是与具体任务相关的。每个程序设计任务都需要重新设计主模块，但子模块是可以重用的。如果子模块在以前项目中曾经开发过，或者可以从市场上购买到，那么就可以直接重用这些子模块，实现快速开发。

程序设计中，子模块是使用某种计算机语言编写出的程序代码。重用子模块实际上是重用其代码，这就是程序设计中的**代码重用**（code reuse）。重用代码的目的是重用该代码所实现的程序功能。代码重用可以减少重复开发，降低开发工作量，同时也可以提高软件质量，因为重用的代码一般都已经过充分测试和运行验证。

代码重用也会影响到大型软件开发的组织与管理方式，例如，现在的软件项目可以重用以前项目所开发的代码（即跨时间段重用）；一个软件项目可以重用另一个项目中的代码（即跨项目重用）；可以重用本单位已有的代码，也可以购买外单位已有的代码，或委托外单位开发所需的子模块（即跨组织机构的重用）。通过购买代码或委托开发可以缩减本单位开发人员的规模，降低项目开发风险。

## 5.1.3　模块的 4 大要素

描述一个算法模块有 4 大要素，分别是**模块名称**、**输入参数**、**返回值**以及**算法**本身。模块的设计者和调用者具有不同的角色，他们对模块及其 4 大要素的理解是不一样的。

### 1. 模块设计者

站在模块设计者角度，他要做的事情是：接收输入参数，将输入参数作为原始数据进行处理，得到计算结果并返回该结果。模块设计者重点考虑处理算法，该算法应能按照模块的功能要求返回正确的计算结果（即返回值）。

模块算法是属于模块设计者的知识产权，模块名称、输入参数和返回值是模块设计者

为他人使用算法所提供的开放接口（统称为模块的调用接口）。调用接口应当是公开的，否则其他人无法使用模块。算法是模块内部的实现细节，可以不对外公开。在程序设计中，只有拿到模块的源代码才能了解算法的实现细节。通常，模块是以编译后的机器语言形式提供的，只能被调用，很难阅读理解，也无法修改。

### 2. 模块调用者

站在模块调用者角度，模块就像是一种函数 $f(x)$，其中 $x$ 是变量。给定某个具体的 $x$ 值就能得到对应的函数值 $f(x)$。程序设计中，模块名称就相当于函数名 $f$；输入参数是原始数据，相当于变量 $x$；模块的计算结果（即返回值）相当于函数值。调用者通过模块名称调用模块，调用时按要求给定具体的输入参数值 $x'$，然后接收返回值，这样就能得到所需要的计算结果。调用者知道了调用接口就可以调用模块，无需了解模块内部的算法细节。

结构化程序设计将一个复杂算法分解成多个模块，其中一个为**主模块**，其他模块统称为**子模块**。主模块负责调用子模块，但反过来子模块不能调用主模块。子模块可以被主模块或其他子模块调用，也可以调用其他子模块。

程序设计是软件开发最重要的阶段。结构化程序设计方法通常以流程图、伪代码或自然语言的形式来描述设计结果，比如各模块的功能、算法以及模块之间的调用关系等，并编写成书面的程序设计报告。程序设计结束后就进入软件开发的下一个阶段——编写程序代码（简称为编码）。编码就是使用某种计算机语言来描述模块的设计结果。编码之前，程序员需要预先选择好编码所用的计算机语言，以及使用该语言所需的集成开发环境。

C++语言支持结构化程序设计方法，以函数的语法形式来描述和组装模块，即函数的定义和调用。

## 本节习题

1. 下列关于结构化程序设计的描述中，错误的是（　　）。
   A. 结构化程序设计将复杂问题分解成若干个模块，分而治之
   B. 所分解出的模块通常要完成某种相对独立的功能
   C. 结构化程序设计通过调用关系将模块组织起来，形成完整的算法流程
   D. 结构化程序设计中各模块之间相互独立，不需要共享数据
2. 结构化程序设计不会影响下列团队开发中的哪项工作？（　　）
   A. 任务分工　　　　B. 程序员之间的协作
   C. 项目组织与管理方式　　　　D. 程序员晋升
3. 一个算法模块包含 4 大要素，下列哪项不属于模块的要素？（　　）
   A. 模块名称　　　　B. 输入参数
   C. 返回值　　　　D. 模块设计者
4. 调用模块的程序员不会关心下列被调模块的哪个要素？（　　）
   A. 模块名称　　　　B. 输入参数
   C. 返回值　　　　D. 算法实现

5. 结构化程序设计方法将程序分解成模块的目的不包括下列哪一项？（　　）
   A. 分工协作　　　　B. 避免重复劳动
   C. 代码重用　　　　D. 美化程序格式

## 5.2　函数的定义和调用

本节以例 5-2 的测算养鱼池工程总造价算法为例，讲解如何使用 C++语言来描述算法模块。C++语言使用**函数**（function）的语法形式来描述模块。描述模块就是定义函数，编写**函数定义**（function definition）代码。例 5-2 的算法共有 3 个模块，其中一个主模块，两个子模块。将每个模块定义成一个函数，共定义 3 个函数。其中描述主模块的称为**主函数**，另外两个描述子模块的称为**子函数**。主函数调用两个子函数分别计算长方形养鱼池、圆形清水池和污水池的造价。定义主函数时需要编写**调用**（call 或 invoke）子函数的语句。

### 5.2.1　函数的定义

**C++语法：定义函数**

```
函数类型 函数名(形式参数列表)
{
    函数体
}
```

语法说明：

- **函数类型**定义函数返回值（即函数值）的数据类型。函数类型由函数功能决定，可以是数组之外的任何数据类型，默认为 int 型。某些函数可能只是完成某种功能，但没有返回值，此时函数类型应定义为 void。
- **函数名**指定函数的名称，由程序员命名，需符合标识符的命名规则。通常函数之间不能重名。
- **形式参数列表**定义了函数接收输入参数所需的变量，这些变量称为**形式参数**，简称为**形参**。可以有多个形参，每个形参以"**数据类型 变量名**"的形式定义，形参之间用","隔开。某些函数不需要输入参数，此时形式参数列表省略为空。
- **函数体**是描述数据处理算法的 C++语句序列，用大括号"{ }"括起来。函数体中可以定义专供本函数使用的变量。如果函数有返回值，则应使用 return 语句返回。返回值的数据类型应与函数类型一致。
- 函数名、形式参数、函数类型和函数体是定义函数时的 4 大要素，它们分别对应了算法模块中的 4 大要素，即模块名称、输入参数、返回值以及算法。其中，"**函数类型 函数名(形式参数列表)**"合起来称为**函数头**，它描述了函数的调用接口。

使用 C++语言将例 5-2 中的两个子模块分别定义成函数。子模块 1 描述了测算长方形养鱼池造价的算法。该算法有两个输入参数（用于接收养鱼池的长和宽），有一个计算结果

（即养鱼池的造价）。将描述子模块 1 的函数命名为 RectCost，例 5-3 给出了该函数的定义代码。

| 例 5-3　测算长方形养鱼池造价模块的函数定义 | | |
|---|---|---|
| **子模块 1**：测算长方形养鱼池造价 | | **子函数 1**：RectCost |
| 1 | 接收主模块传递过来的两个原始数据，即长方形养鱼池的长和宽。 | double RectCost(double a, double b)<br>{ |
| 2 | 再定义一个变量，申请保存造价计算结果所需的内存空间。 | double cost ; |
| 3 | 按照公式计算造价：造价=长*宽*单价。 | cost = a * b * 10 ; |
| 4 | 将计算结果返回给主模块。 | return cost ;<br>} |

子模块 2 描述了测算圆形水池造价的算法。该算法有一个输入参数（用于接收水池的半径），有一个计算结果（即水池的造价）。将描述子模块 2 的函数命名为 CircleCost，例 5-4 给出了该函数的定义代码。

| 例 5-4　测算圆形水池造价模块的函数定义 | | |
|---|---|---|
| **子模块 1**：测算圆形水池造价 | | **子函数 1**：CircleCost |
| 1 | 接收主模块传递过来的一个原始数据，即圆形水池的半径。 | double CircleCost(double r)<br>{ |
| 2 | 再定义一个变量，申请保存造价计算结果所需的内存空间。 | double cost ; |
| 3 | 按照公式计算造价：<br>造价=3.14*半径*半径*单价。 | cost = 3.14 * r * r * 10 ;<br>return cost ; |
| 4 | 将计算结果返回给主模块。 | } |

主模块负责输入原始数据（包括养鱼池的长和宽、清水池和污水池的半径），计算并显示最终的工程总造价。主模块在计算工程总造价时分别调用了子模块 1 和子模块 2，其中子模块 2 被调用了两次。C++语言使用函数调用语句来描述模块的调用，因此在定义主函数之前需先讲解一下调用函数的语法。

## 5.2.2　函数的调用

**C++语法：调用函数**

```
函数名(实际参数列表)
```

语法说明：

- **函数名**指定被调用函数的名称。一个函数调用另一个函数，调用别人的函数称为**主调函数**，被调用的函数称为**被调函数**。

- **实际参数列表**给出主调函数传递给被调函数的实际参数值，简称为**实参**。实参可以是常量、变量或表达式，参数之间用“,”隔开。调用时自动按位置顺序将实参一一赋值给对应的形参，这称为函数调用时的**参数传递**。调用函数时的实参应当与被调函数定义中的形参个数一致，类型一致。
- **“函数名(实际参数列表)”** 就是在调用某个函数。有返回值的函数调用可作为操作数参与表达式运算，该操作数等于函数的返回值。某些函数可能只是完成某种功能，但没有返回值。无返回值的函数调用加分号“;”，即构成一条函数调用语句。

一个 C++程序由一个主函数和若干子函数组成，主函数、子函数分别对应程序设计阶段的主模块和子模块。主函数负责调用子函数，子函数可以再调用其他子函数。C++语言规定程序的主函数名为 main，通常不接收输入参数，返回值为 int 型。使用 C++语言描述例 5-2 中的主模块，所定义的主函数代码见例 5-5。

**例 5-5　测算养鱼池工程总造价主模块的函数定义**

| 主模块：测算养鱼池工程总造价 | 主函数：main |
|---|---|
| 1 定义变量，申请保存原始数据（包括养鱼池的长和宽、清水池和污水池的半径）和计算结果（工程总造价）所需的内存空间。<br>2 从键盘输入原始数据，包括养鱼池的长和宽、清水池和污水池的半径。<br>3 **调用子模块** 1：跳转到子模块 1，跳转前将长方形养鱼池的长和宽的数据传递过去，然后将子模块 1 返回的计算结果累加到工程总造价。<br>4 **调用子模块** 2：跳转到子模块 2，跳转前将圆形清水池的半径传递过去，然后将子模块 2 返回的计算结果累加到工程总造价。<br>5 **调用子模块** 2：跳转到子模块 2，跳转前将圆形污水池的半径传递过去，然后将子模块 2 返回的计算结果累加到工程总造价。<br>6 在显示器上显示测算出的工程总造价。 | int main( )<br>{<br>    double length, width;<br>    double r1, r2;<br>    double totalCost = 0;<br>    cout << "请输入长方形的长和宽：" ;<br>    cin >> length >> width;<br>    cout << "请输入清水池和污水池的半径：" ;<br>    cin >> r1 >> r2;<br><br>    totalCost += RectCost( length, width );<br>    totalCost += CircleCost( r1 );<br>    totalCost += CircleCost( r2 );<br><br>    cout << "工程总造价为 " << totalCost << endl;<br>    return 0;<br>} |

例 5-5 中的主函数包含 3 条调用函数语句。

（1）totalCost += RectCost( length, width );

这条语句调用子函数 RectCost 计算长方形养鱼池的造价。调用时将保存在变量 length

和 width 中的长宽数据作为实参值传递给函数 RectCost 中对应的形参 a 和 b，然后等待接收函数的返回值，即养鱼池造价，将其累加到变量 totalCost 上。

（2）totalCost += CircleCost( r1 );

这条语句调用子函数 CircleCost 计算圆形清水池的造价。调用时将保存在变量 r1 中的清水池半径作为实参值传递给函数CircleCost中对应的形参r，然后等待接收函数的返回值，即清水池造价，将其累加到变量 totalCost 上。

（3）totalCost += CircleCost( r2 );

这条语句再次调用子函数 CircleCost 计算圆形污水池的造价。调用时所传递的实参值改为污水池的半径 r2，相应地所接收到的函数返回值就是污水池造价，将其累加到变量 totalCost 上。

子函数 CircleCost 被主函数调用了两次，我们称函数 CircleCost 的代码被重用了。重用时函数的代码相同，但所传递的实参值不同（分别是 r1 和 r2），因此得到了不同的返回值（分别对应清水池和污水池造价）。函数的代码重用，重用的是算法，而算法所处理的数据通常是不同的。将数据提炼出来定义成变量，这种做法称为**数据参数化**。函数的形式参数就是将所处理的数据经过提炼而形成的变量，它是提高函数代码重用性、扩大重用范围的重要手段。请比较下面两个计算圆面积的函数。

| **带参数**的计算圆面积函数 | **不带参数**的计算圆面积函数 |
| --- | --- |
| double CircleArea1(double r)<br>{<br>    double area;<br>    area = 3.14 * r * r; // 计算半径为r的圆面积<br>    return area;<br>} | double CircleArea2( )<br>{<br>    double area;<br>    area = 3.14 * 5 * 5;  // 计算半径为 5 的圆面积<br>    return area;<br>} |

函数 CircleArea1 将求圆面积算法中的半径提炼成形式参数 r，用于接收不同的半径值。其他函数在调用 CircleArea1 时，只要传递不同的实参值就可以得到不同半径的圆面积。例如：

```
cout << CircleArea1( 5 );          // 得到半径为 5 的圆面积
cout << CircleArea1( 10 );         // 得到半径为 10 的圆面积
```

而函数 CircleArea2 没有形参，计算圆面积时将半径写成常量 5。该函数只能计算半径为 5 的圆面积，其重用性大大降低，重用范围非常窄。

程序员在定义函数时，可以利用数据参数化将算法中的数据提炼出来，将它们定义成形式参数，这样就能有效提高函数代码的重用性和重用范围。

### 5.2.3 函数应用举例

本节我们通过一个具体的程序设计任务来复习函数的定义和调用。假设有某位程序员甲要编写一个将摄氏温度转换成华氏温度的 C++程序。可是程序员甲不知道换算公式，他只能按如下的方式来编写主函数代码。

```
#include <iostream>
using namespace std;
int main( )
{
    double ctemp, ftemp;            // 申请内存空间
    cin >> ctemp;                   // 从键盘输入摄氏温度
    ftemp = … ;                     // 温度换算，用省略号代替换算公式
    cout << ftemp;                  // 在显示器上输出华氏温度
    return 0;                       // 程序结束，返回操作系统
}
```

恰好有另外一位程序员乙知道如何进行温度换算。那么程序员乙该如何编写温度换算函数呢？编写函数时，程序员乙应当依据函数定义时的 4 大要素来思考问题。

（1）函数名称。应当起一个简单好记的名字，于是程序员乙将函数命名为 C2F（即摄氏温度到华氏温度）。

（2）输入参数。为了将摄氏温度转换成华氏温度，程序员甲调用函数 C2F 时应当传递一个摄氏温度值，因此程序员乙需要定义一个形参来接收这个值。考虑到这个数值可能是实数，于是程序员乙将形参定义为：double c，其中 c 是形参名。

（3）函数类型。用 double 型摄氏温度计算得到的华氏温度当然也是 double 型的，因此程序员乙将函数 C2F 的返回值类型（即函数类型）定义为 double。

（4）函数体。实现温度换算算法对程序员乙来说是小菜一碟，用 C++语言将温度换算公式编写成表达式就可以了。

程序员乙按照 C++语言的语法规则编写出如下温度换算函数 C2F 的定义代码：

```
double C2F( double c )              // 函数头：函数类型 函数名(形式参数)
{           // 大括号中由 3 条 C++语句构成的序列就是实现温度换算算法的函数体
    double f;                       // 先定义一个保存华氏温度的变量 f
    f = c * 1.8 + 32;               // 温度换算公式
    return f;                       // 返回结果 f
}
```

有了 C2F 函数，程序员甲将主函数中被省略的换算公式改为对函数 C2F 的调用，其调用形式如下：

```
ftemp = C2F( ctemp );               // 调用函数 C2F 进行温度换算
```

调用时，程序员甲将保存在变量 ctemp 中的摄氏温度值（它是之前用 cin 指令从键盘输入的）传递给函数 C2F。ctemp 被称为是调用函数时的实参。那么将 C2F( ctemp )赋值给 ftemp 是干什么，C2F( ctemp )具体是什么意思呢？

对比一下我们所熟悉的正弦函数 sin($x$)，其中 sin 是函数名，$x$ 是角度，sin($x$)就是对应的函数值（即角度 $x$ 对应的正弦值）。在 C2F( ctemp )中，C2F 是函数名，ctemp 是摄氏温度，C2F( ctemp )就是对应的函数值（即摄氏温度 ctemp 对应的华氏温度值）。将 C2F( ctemp )赋值给 ftemp，就是将摄氏温度 ctemp 对应的华氏温度值赋值给变量 ftemp。与正弦函数 sin($x$)相比，温度换算函数 C2F 的计算方法是由程序员乙用 C++语言定义出来的。

## 5.2.4 函数的执行

本节仍以测算养鱼池工程总造价为例，具体讲解计算机如何执行带子函数的C++程序。

将描述例 5-2 算法模块的 3 个函数定义代码（例 5-3~5-5）编写在同一个源程序文件（扩展名为.cpp）中，这就构成了一个完整的测算养鱼池工程总造价的 C++程序（例 5-6）。为了使用标准输入流 cin 和标准输出流 cout 来输入/输出数据，需要在程序头部增加两条导入外部程序的语句。将该 C++源程序编译、连接，生成一个可执行程序。执行该程序就可以输入相关数据，测算出养鱼池的工程总造价。

**例 5-6 测算养鱼池工程总造价的 C++程序**

```
#include <iostream>                    // 导入外部程序：标准输入/输出流
using namespace std;

double RectCost(double a, double b)    // 计算长方形养鱼池造价的函数定义
{
    double cost ;
    cost = a * b * 10 ;
    return cost ;
}

double CircleCost(double r)            // 计算圆形水池造价的函数定义
{
    double cost ;
    cost = 3.14 * r * r * 10 ;
    return cost ;
}

int main( )                            // 主函数定义
{
    double length, width;              // 定义变量：分别保存长方形养鱼池的长和宽
    double r1, r2;                     // 定义变量：分别保存圆形清水池和污水池的半径
    double totalCost = 0;              // 定义变量：保存最终的计算结果，即总造价
    cout << "请输入长方形的长和宽：" ;
    cin >> length >> width;
    cout << "请输入清水池和污水池的半径：" ;
    cin >> r1 >> r2;

    totalCost += RectCost( length, width );  // 调用函数 RectCost 计算长方形养鱼池造价
    totalCost += CircleCost( r1 );     // 调用函数 CircleCost 计算圆形清水池造价
    totalCost += CircleCost( r2 );     // 再次调用函数 CircleCost 计算圆形污水池造价

    cout << "工程总造价为 " << totalCost << endl;
    return 0;
}
```

一个 C++程序必须有且只能有一个名为 main 的主函数，可以没有子函数，也可以包含多个子函数。计算机执行程序是从主函数的第一条语句开始执行的，一直执行到最后一条语句结束，或执行到主函数中的 return 语句时中途退出。

如果主函数调用了某个子函数，计算机在执行到函数调用语句时将暂停主函数的执行，跳转去执行子函数中的代码。执行完子函数或执行到其中的 return 语句时，退出子函数，返回主函数继续执行其中剩余的指令。子函数还可以调用别的子函数，这就形成了函数的**嵌套调用**。图 5-5 展示了例 5-6 程序中函数的执行过程。

图 5-5 中函数执行过程的说明如下：

① 计算机执行程序时，从主函数 main 的第一条语句开始顺序执行。

② 当执行到代码第 12 行调用函数 RectCost 的语句时，计算机将暂停主函数 main 的执行，跳转去执行子函数 RectCost。跳转前，将保存在变量 length 和 width 中的长和宽数据作为实参值，按位置顺序一一赋值给函数 RectCost 中对应的形参 a 和 b，这就是函数调用时的参数传递。

③ RectCost 的函数体是实现求长方形养鱼池造价算法的语句序列。它根据形参 a、b 所接收到的长宽值计算长方形养鱼池造价，计算结果保存在变量 cost 中。

④ 当执行到语句“return cost;”时返回计算结果（即 cost 中的数值），然后退出子函数 RectCost，返回主函数 main。主函数接收返回值，即长方形养鱼池造价，将其累加到变量 totalCost 上。

⑤ 计算机继续执行主函数中剩余的语句。

⑥~⑨ 当执行到代码第 15 行调用函数 CircleCost 的语句时，计算机再次暂停主函数 main 的执行，跳转去执行子函数 CircleCost，执行过程与 RectCost 类似。执行 CircleCost 时只传递一个参数，即清水池的半径 r1，返回值是圆形清水池的造价。

⑩~⑫ 当执行到代码第 18 行再次调用函数 CircleCost 的语句时，计算机又一次暂停主函数 main 的执行，跳转去执行子函数 CircleCost。同一段 CircleCost 函数代码被执行了两次，所不同的是第 2 次执行时传递的实参值是污水池的半径 r2，相应的其返回值就是污水池的造价。

⑬ 计算机继续执行主函数中剩余的语句（第 20 和 21 行），显示养鱼池工程总造价，然后退出主函数。至此，测算养鱼池工程总造价程序的执行就全部结束了，退出程序，返回操作系统。

计算机执行函数调用语句时，主调函数与被调函数之间有两次数据传递：第一次是将主调函数中的实际参数按照位置顺序一一赋值给被调函数中对应的形式参数；第二次是被调函数使用 return 语句将计算结果返回给主调函数，主调函数接收返回值。如果一个函数没有形式参数，则称为**无参函数**。调用无参函数时没有参数传递。一个函数也可以没有返回值，此时该函数的函数类型应定义为 void，调用无返回值函数时不传递返回值。

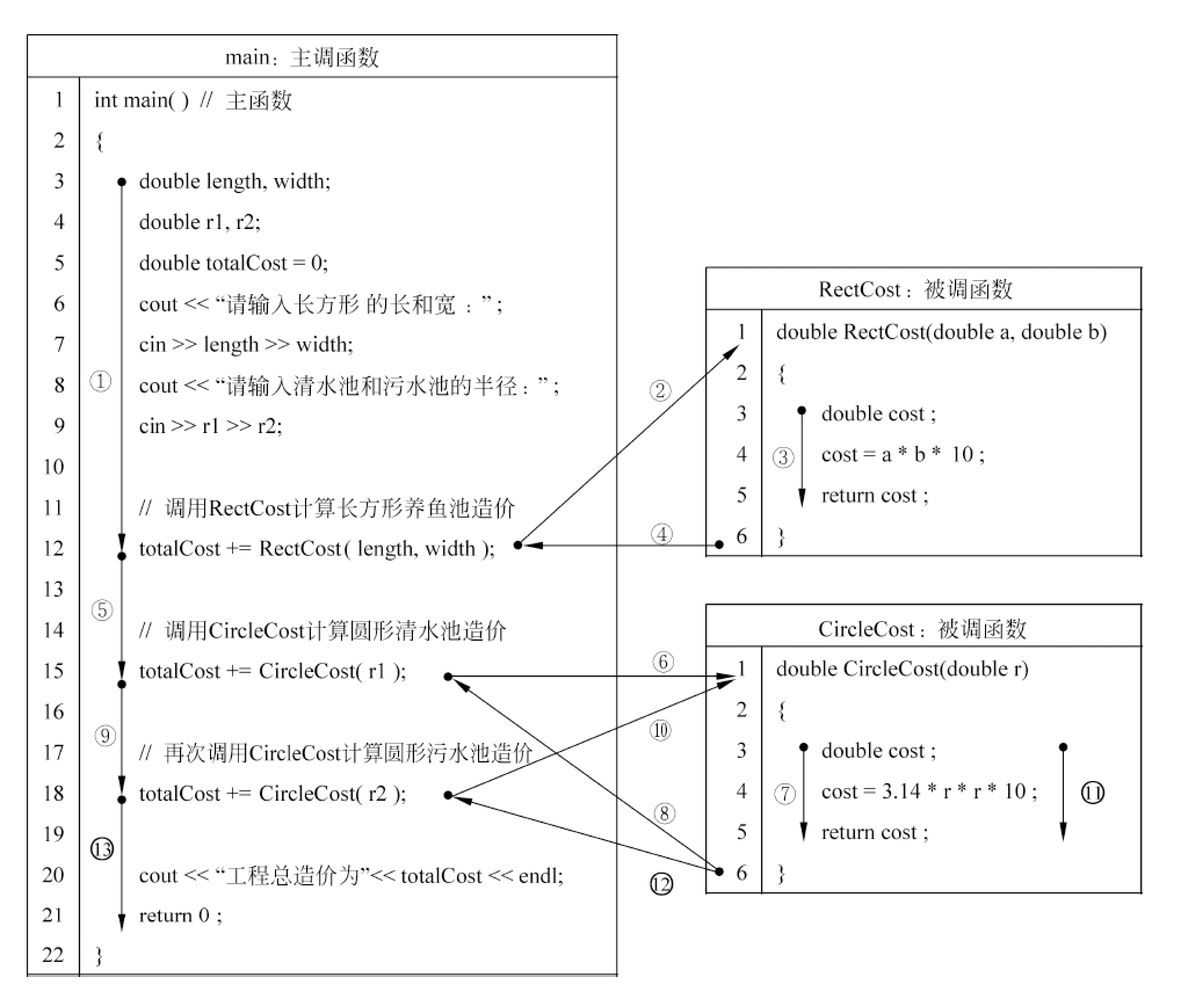



图 5-5　例 5-6 程序中函数的执行过程

### C++语法：return 语句

```
return (表达式) ;
或
return ;
```

语法说明：

- 如果函数有返回值，则应当使用“return (表达式) ;”语句结束函数执行，返回主调函数。**表达式**指定返回时的返回值，其数据类型应当与函数头中的函数类型一致。常量或变量可认为是一个最简单的表达式。**小括号**“( )”可以省略；
- 如果函数的类型为 void，即没有返回值，则可以使用“**return** ;”语句结束函数执行，返回主调函数。如果省略“return ;”语句，则在执行完函数体中最后一条语句后自动返回主调函数。

在 return 语句中使用表达式，可以简化例 5-6 中函数 RectCost 和 CircleCost 的代码。例 5-7 给出了简化后的代码。

| 例 5-7　在 return 语句中使用表达式简化函数代码 | |
|---|---|
| 函数 RectCost：简化前 | 函数 RectCost：简化后 |
| double RectCost(double a, double b)<br>{<br>    double cost ;<br>    cost = a * b * 10 ;<br>    return cost ;<br>} | double RectCost(double a, double b)<br>{<br>    return (a * b * 10) ;<br>} |
| 函数 CircleCost：简化前 | 函数 CircleCost：简化后 |
| double CircleCost(double r)<br>{<br>    double cost ;<br>    cost = 3.14 * r * r * 10 ;<br>    return cost ;<br>} | double RectCost(double r)<br>{<br>    return (3.14 * r * r * 10) ;<br>} |

本节最后提一个问题：程序员在源程序中定义的子函数一定会被计算机执行吗？

这个问题的答案是：

（1） 源程序中定义的子函数可能执行，也可能不执行。只有被主函数直接或间接调用的子函数才会被执行，否则就不会被执行。

（2） 源程序中定义的子函数可能被执行多次。子函数被调用一次就执行一次，调用多少次则执行多少次。

## 5.2.5　函数的声明

C++源程序中，变量遵循“先定义，后访问”的原则，函数则应遵循“**先定义，后调用**” 的原则。例 5-6 中，主函数 main 调用两个子函数 RectCost 和 CircleCost，因此编写程序代码时将这两个子函数定义在了主函数的前面，这样主函数中的函数调用语句就满足了“先定义，后调用”的要求。

C++语言还规定，只要对被调函数的函数原型进行**声明**（declaration），就可以调用该函数，而其函数定义可以放在后面，即“**先声明，后调用**”。**函数原型**（prototype）就是函数的调用接口，其中包括函数名称、形式参数列表和函数类型，但不包括函数体。函数原型也被称为函数的签名（signature）。

**C++语法：声明函数原型**

| 函数类型　函数名(形式参数列表) ; |
|---|
| 语法说明：<br>■ 一个函数的原型声明语句可简单认为是由该函数定义中的**函数头**加分号“;”组成。<br>■ 形式参数列表中各形参的数据类型是函数原型声明中必须包含的信息。它除了指明形参的数据类型之外，还暗含了形参的个数。而形参变量名不重要，可以省略。声明形参名的作用是为了便于调用该函数的程序员理解参数的含义。 |

- C++源程序中，被调函数的原型声明语句可放在主调函数定义之前，或整个源程序文件的开头，也可以放在主调函数的函数体中，但必须在函数的调用语句之前。
- 如果被调函数与主调函数定义在同一个源程序文件中，并且被调函数定义在主调函数之前，则被调函数的原型声明语句可以省略。
- 声明函数原型的目的是将被调函数的调用接口预先告知编译器程序，这样编译器就可以按照该函数调用接口来检查其后续的调用语句是否正确。

举例：例 5-6 中函数 RectCost 的原型声明

```
double RectCost(double a, double b) ;      // 复制 RectCost 函数定义中的函数头，再加“;”
```

或

```
double RectCost(double, double) ;          // 省略上述原型声明中的形参变量名
```

如果改变例 5-6 中 3 个函数定义代码的位置顺序，将子函数 RectCost 和 CircleCost 移到主函数 main 的后面，则需要在主函数前声明这两个子函数的原型（见例 5-8）。

**例 5-8　测算养鱼池工程总造价的 C++程序（声明函数原型）**

```
#include <iostream>
using namespace std;

double RectCost(double a, double b);            // 声明子函数 RectCost 的原型
double CircleCost(double r);                    // 声明子函数 CircleCost 的原型

int main( )                                     // 主函数定义（略）
{
    …
    totalCost += RectCost( length, width );     // 调用函数 RectCost 计算长方形养鱼池造价
    totalCost += CircleCost( r1 );          // 调用函数 CircleCost 计算圆形清水池造价
    totalCost += CircleCost( r2 );          // 再次调用函数 CircleCost 计算圆形污水池造价
    …
}

double RectCost(double a, double b)         // 计算长方形养鱼池造价的函数定义（省略）
{  …  }

double CircleCost(double r)                 // 计算圆形水池造价的函数定义（省略）
{  …  }
```

## 5.2.6　程序员与函数

本节最后再简单描述一下程序员与函数的关系。在结构化程序设计中，程序员通常是将相对独立并广泛使用的算法提炼出来，编写成函数形式的可重用代码。

求 $x$ 的 $n$ 次方 $x^n$ 是一种常用的算法，假设 $x$ 为实数，$n$ 为整数。程序员可以将这个算法模块编写成一个函数，那么该如何用 C++语言定义这个函数呢？回忆一下描述算法模块的 4 大要素，它们分别是模块名称、输入参数、返回值和算法。

（1）**模块名称**。假定求 $x^n$ 的算法模块已被命名为 power（幂），程序员在定义函数时

可以将函数名简写成 pow。

（2）**输入参数**。该算法需要什么输入参数呢？应当需要一个实数 $x$ 和一个整数 $n$。程序员定义函数时需要增加两个形参来接收输入参数。假设程序员增加如下两个形参：

（double x, int n）

（3）**返回值**。算法处理的结果就是返回值。定义函数时，函数的数据类型就是返回值的数据类型。求 $x^n$ 算法的结果是一个实数，程序员应当将函数 pow 的数据类型定义成某个实数类型。因为上一步已将 $x$ 定义成 double 类型，相应地这里的函数类型也应当定义成 double 类型。

（4）**算法**。通过循环结构可以实现求 $x^n$ 的算法。程序员编写代码时应当注意算法细节，例如指数 $n$ 为 0 或负数时该怎么办？

例 5-9 给出一个定义 pow 函数的示例代码。pow 函数的功能是求 $x$ 的 $n$ 次方 $x^n$。

**例 5-9　定义 pow 函数（求 $x$ 的 $n$ 次方 $x^n$）的示例代码**

```
double pow(double x, int n)            // 函数头中依次定义函数类型、函数名和形参列表
{                                      // 在函数体中编写实现算法的 C++代码
    if (x == 0)   return 0;            // x 为 0 时直接返回 0
    if (n == 0)   return 1;            // n 为 0 时直接返回 1
    int loops;                         // 定义保存循环次数的变量 loops
    if (n<0)   loops= -n;
    else   loops= n;
    double result = 1;                 // 定义保存结果的变量 result，初始化为 1
    for (int m = 1; m <= loops; m++)   // 通过循环结构可以实现求 x 的 n 次方的算法
    {
        if (n>0)   result *= x;
        else   result /= x;
    }
    return result;                     // 返回结果
}
```

pow 函数定义好之后，凡是需要求幂的地方都可以直接调用这个函数，不需要再重复编写代码。使用函数可以有效地减少程序中的重复代码。假设有 3 个程序员甲、乙、丙，他们在编写程序时都需要用到求幂的功能。

（1）程序员甲。需要求 $3.5^2$，则可以用如下的形式来调用函数 pow：

```
cout << pow(3.5, 2) << endl;
```

其中，3.5 和 2 是调用函数 pow 时程序员甲给出的实参。编写函数调用语句时，程序员应按照被调函数的要求传递指定个数和类型的实参。

（2）程序员乙。需要求 $8.1^{-3}$，则可以用如下的形式来调用函数 pow：

```
cout << pow(8.1, -3) << endl;
```

可以看出，调用函数时程序员传递不同的实参值就会得到不同的返回值。

（3）程序员丙。是个好心人，他为读者提供了如下调用 pow 函数的完整示例代码。

```
#include <iostream>
using namespace std;
```

```
double pow(double x, int n);   // 声明函数 pow 的原型，即调用接口
int main( )
{
    double x; int y;      cin >> x >> y;   // 从键盘输入两个数值
    cout << pow(x, y) << endl;   // 调用函数 pow 求 x 的 y 次方
    return 0;
}
```

编写好的函数可以被同一项目组中的所有程序员调用，也可以在今后的项目中继续使用，或者是以公开销售的形式提供给任何其他程序员使用。

## 本节习题

1. 一个 C++程序必须有且只有一个名为（　　）的主函数。

A. function　　B. main　　C. Main　　D. MAIN

2. C++语言中的函数是实现某种功能的独立代码段。一个 C++程序应包含（　　）函数。

A. 至少 1 个　　B. 至少 2 个
C. 至少 3 个　　D. 可包含任意多个

3. 下列关于 C++函数的叙述，正确的是（　　）。

A. C++程序总是从源程序中第一个定义的函数开始执行
B. C++程序总是从 main 函数开始执行
C. C++程序中被调用的子函数必须定义在 main 函数之前
D. C++程序中的 main 函数必须放在程序的开始部分

4. 定义函数时，如果函数没有返回值，则应将函数类型指定为（　　）。

A. int　　B. bool　　C. void　　D. null

5. 如需定义一个求圆面积的函数 Area，下列哪个函数定义是正确的？（　　）

A. double Area(double r)　{　double s;　s = 3.14*r*r;　return s;　}
B. void Area(double r)　{　double s;　s = 3.14*r*r;　return s;　}
C. double Area( )　{　double s;　s = 3.14*r*r;　return s;　}
D. double Area(double r)　{　Area = 3.14*r*r;　}

6. 定义函数“int * fun( ) { … }”，该函数返回值的类型是（　　）。

A. int　　B. int *　　C. bool　　D. 语法错误

7. 定义函数“double fun( ) { … }”，下列调用正确的语句是（　　）。

A. int x = fun( );　　B. float x = fun( );
C. double x = fun( );　　D. double x = fun(3.5);

8. 定义函数“int fun(int x) { … }”，下列调用不正确的语句是（　　）。

A. int y = fun('5');　　B. int y = fun(5);
C. int x = fun( fun(5) );　　D. int x = fun(“5”);

9. 定义函数“int fun(int x) { … }”，下列对该函数的声明语句中语法错误的是（　　）。

A. int fun(int);　　B. int fun(int x);
C. int fun(int y);　　D. void fun(int x);

10. 已知函数调用语句“char c = fun('A', 5.5);”，则该函数定义的函数头最有可能是（　　）。

A. void fun(char x, double y)　　B. char function(char a, double b)

C. char fun(char c, double d)　　D. char fun(double x, char y)

# 5.3　数据的管理策略

结构化程序设计在将一个数据处理过程分解成多个算法模块之后，模块之间需要共享数据。C++语言以函数的语法形式来描述模块，每个模块被定义成一个函数。函数之间需要共享数据才能完成规定的数据处理任务。C++语言为程序员提供了两种数据管理策略，分别是分散管理和集中管理。

**数据分散管理**策略就是将数据分散交由各个函数管理。函数各自定义变量申请自己所需的内存空间，其他函数不能直接访问其中的数据，需要时可通过数据传递来实现共享。采用分散管理策略时，程序员应当将定义变量语句放在函数的函数体中，这样所定义的变量称为**局部变量**（local variable）。局部变量属本函数所有，其他函数不能直接访问。

**数据集中管理**策略就是将数据集中管理，统一定义公共的变量来存放共享数据，所有函数都可以访问。采用集中管理策略时，程序员应当将定义变量语句放在函数外面（不在任何函数的函数体中），这样所定义的变量称为**全局变量**（global variable）。全局变量不属于任何函数，是公共的，所有函数都可以访问。

## 5.3.1　数据分散管理，按需传递

例 5-6 所示的测算养鱼池工程总造价程序采用的就是数据分散管理策略。它将程序所涉及的原始数据、中间结果和最终结果分散交由 3 个函数进行管理。

主函数 main 负责管理原始数据和最终结果。主函数 main 首先定义变量申请所需的内存空间（代码 20~22 行），然后从键盘输入原始数据（代码 23~26 行），通过调用子函数 RectCost 和 CircleCost 分别计算长方形养鱼池、圆形清水池和污水池的造价，并累加到变量 totalCost 上，最后显示工程总造价。

主函数 main 所定义的变量是局部变量，其他函数不能直接访问。例如，函数 RectCost 在计算长方形养鱼池造价时需要共享主函数中的长宽数据，这两个数据存放在变量 length 和 width 中。RectCost 不能直接访问主函数中的变量，无法读取其中的数据，该怎么办呢？另外，RectCost 所计算出的长方形养鱼池造价是一个中间结果，存放在变量 cost 中。变量 cost 是 RectCost 定义的局部变量，主函数也不能直接访问，无法获得计算结果，这又该怎么办呢？采用分散管理策略时，主调函数和被调函数不能互相访问对方的局部变量，无法共享数据。

C++语言通过**形实结合**和**返回值**这两个数据传递机制实现了主调函数和被调函数间的数据共享。主调函数的函数体中含有对被调函数的调用语句，当执行该调用语句时，计算机会自动执行两次数据传递操作。

### 1. 形实结合

当执行到函数调用语句时，计算机将暂停主调函数的执行，跳转去执行被调函数。跳转前，计算机自动将调用语句中的实参值按照位置顺序，一一赋值给被调函数中对应的形式参数，这就是**形实结合**。形实结合的作用是将主调函数中的原始数据传递给被调函数，被调函数再对形参所接收的数据进行处理。

### 2. 返回值

计算机跳转到被调函数后，开始执行被调函数的函数体。当执行到 return 语句时，首先计算语句中的表达式，然后将结果返回给主调函数，这个结果被称为是被调函数的返回值。返回值的作用是将被调函数的计算结果传回主调函数。

程序员应当知道，形实结合和返回值是函数调用语句所暗含的操作。如果一个函数没有形式参数，即无参函数，则调用无参函数没有形实结合。一个函数也可以没有返回值，即函数类型为 void，调用无返回值函数不传递返回值。

## 5.3.2 数据集中管理，全局共享

数据集中管理策略就是将数据集中管理，统一定义公共的变量来存放共享数据，所有函数都可以访问。采用集中管理策略时，程序员应当将定义变量语句放在函数外面（不在任何函数的函数体中），这样所定义的变量称为全局变量。全局变量不属于任何函数，是公共的，所有函数都可以访问。采用数据集中管理策略可以避免频繁地在函数间传递数据。例 5-10 就采用数据集中管理策略，重新编写例 5-6 中的测算养鱼池工程总造价程序。

**例 5-10 测算养鱼池工程总造价的 C++程序（数据集中管理策略）**

```
#include <iostream>
using namespace std;

// 下列变量被定义在函数外面，是全局变量，所有函数都可以访问
double length, width;              // 定义变量：分别保存长方形养鱼池的长和宽
double r1, r2;                     // 定义变量：分别保存圆形清水池和污水池的半径
double totalCost = 0;              // 定义变量：保存最终的计算结果，即总造价

void RectCost( )           // 计算长方形养鱼池造价：改用全局变量后函数无形参、无返回值
{
    double cost ;
    cost = length * width * 10 ;   // 直接读取全局变量 length 和 width 中的长宽数据
    totalCost += cost ;            // 将计算结果直接累加到全局变量 totalCost 中
    return;                        // 无返回值，该语句可以省略
}

double CircleCost(double r)        // 计算圆形水池造价：改用全局变量后函数定义没有改变
{
    double cost ;
    cost = 3.14 * r * r * 10 ;
    return cost ;
```

```
22  }
23  
24  int main( )                              // 主函数定义
25  {
26      // 将例 5-6 在此定义的局部变量全部移到函数外面（第 5~7 行），变成全局变量
27      // 下列语句将键盘输入的原始数据直接存放到各全局变量中
28      cout << "请输入长方形的长和宽：" ;
29      cin >> length >> width;
30      cout << "请输入清水池和污水池的半径：" ;
31      cin >> r1 >> r2;
32  
33      RectCost( );                         // 调用函数 RectCost 计算长方形养鱼池造价
34      totalCost += CircleCost( r1 );       // 调用函数 CircleCost 计算圆形清水池造价
35      totalCost += CircleCost( r2 );       // 再次调用函数 CircleCost 计算圆形污水池造价
36  
37      cout << "工程总造价为 " << totalCost << endl;
38      return 0;
39  }
```

例 5-10 的程序说明如下：

（1）将局部变量改为全局变量（代码第 5~7 行）。将例 5-6 中主函数中定义的局部变量全部移到函数外面定义，变成全局变量。这些变量是公共的，所有函数都可以访问。

（2）修改函数 RectCost 的定义（代码第 9~15 行）。函数 RectCost 的功能是计算长方形养鱼池造价。改用全局变量后，RectCost 不再需要定义形式参数接收主函数传递过来的长宽数据，而是直接从全局变量 length 和 width 中读取。计算所得到的养鱼池造价也不再需要通过返回值返回主函数，而是直接累加到全局变量 totalCost 上。因为没有返回值，函数类型需改为 void，语句“return;”也不包含任何表达式。本例中的这条 return 语句可以省略，因为它是函数的最后一条语句。省略时，计算机会在执行完函数体中的代码后自动退出，返回主函数。

函数中的长方形养鱼池造价，它是一个中间结果。在累加到全局变量 totalCost 上之后就没用了，其他函数也不需要共享这个数据。因此函数 RectCost 定义局部变量 cost 来存放这个中间结果，只供本函数访问。

通常，一个 C++程序既有需全局共享的数据，也有仅供局部使用的数据。程序员应合理地选择管理策略：为需要全局共享的数据定义全局变量，集中管理，供所有函数访问，这样可以有效减少函数间的数据传递；为局部使用的数据定义局部变量，分散交由各个函数自己管理，这样可以降低管理的复杂性。

（3）函数 CircleCost 的定义保持不变（代码第 17~22 行）。为什么改用全局变量后，函数 CircleCost 的定义保持不变，继续保留形参和返回值呢？函数 CircleCost 的功能是计算圆形清水池和圆形污水池的造价。计算这两个水池造价的算法完全一样，只是半径不同。因此主函数在计算两个水池造价时都调用函数 CircleCost，但传递了不同的实参值（代码第 34 和 35 行）。

可以看出，CircleCost 是被重用的函数。重用函数，重用的是算法，但通过形式参数可以接收不同的实参值，得到不同的计算结果。即使改用全局变量，函数 CircleCost 仍需保留形式参数和返回值，这样可以保证其重用性。

（4）修改主函数 main 的定义（代码第 24~39 行）。

改用全局变量后，主函数不再定义任何局部变量，从键盘输入的原始数据被直接存入各全局变量。另外，读者还需要关注其中 3 条函数调用语句的变化。

■ 代码第 33 行。

```
RectCost( );                          // 调用函数 RectCost 计算长方形养鱼池造价
```

主函数调用函数 RectCost 来计算长方形养鱼池造价。改用全局变量后，函数 RectCost 不需要主函数传递原始数据，而是直接从全局变量 length 和 width 中读取长宽值。计算所得到的养鱼池造价被直接累加到全局变量 totalCost 上，不需要返回给主函数，没有返回值。因此主函数在调用该函数时不需要给实参，也不接收返回值，直接加分号“;”即构成一条完整的函数调用语句。

■ 代码第 34 和 35 行。

```
totalCost += CircleCost( r1 );        // 调用函数 CircleCost 计算圆形清水池造价
totalCost += CircleCost( r2 );        // 再次调用函数 CircleCost 计算圆形污水池造价
```

主函数两次调用函数 CircleCost，分别计算圆形清水池和圆形污水池的造价。改用全局变量后，主函数调用 CircleCost 的形式表面看起来与例 5-6 没有什么变化，但请注意：此时传递的实参值是全局变量 r1、r2 的值，接收到的返回值也是累加到全局变量 totalCost 上。

## 5.3.3 变量的作用域

假设甲乙两位程序员合作编写某个程序。双方约定：先定义公共的全局变量来存放共享数据，然后分头编写各自的子函数，其示意代码如下。

```
int x = 10;                           // 定义全局变量 x 存放共享数据 10
void fun1( )                          // 程序员甲负责编写子函数 fun1
{
    int y;                            // 程序员甲定义一个局部变量 y
    y = 100;                          // 将 y 赋值为 100
    …
}
void fun2( )                          // 程序员乙负责编写子函数 fun2
{
    int y;                            // 程序员乙也定义了一个局部变量 y
    y = 200;                          // 将 y 赋值为 200
    …
}
```

在上述示意代码中，程序员甲乙两人各自定义一个局部变量 y，并分别赋值为 100 和 200。请读者考虑这样一个问题：这两个局部变量可以重名吗？如果不能重名，那么程序员

需要经常协商才能避免相互间的重名问题，这听起来很麻烦。如果可以重名，那么程序员对自己变量赋值，这会不会影响到对方的重名变量？另外，程序员在函数中定义的局部变量能与全局变量重名吗？

为了规范变量重名问题，C++语言将不同变量的使用范围限定在某个特定的代码区域内，这就是变量作用域的概念。例如，定义在函数内部的变量是局部变量，只能被本函数访问；定义在函数外面的变量是全局变量，可以被所有函数访问。划定变量的作用域范围，可以避免不同程序员所定义变量之间相互干扰。本节将具体介绍 C++语言中变量的类型及其作用域。

#### 1. 变量类型及其作用域

在 C++源程序中，变量需要遵循“先定义，后访问”的原则，即变量在定义之后，其后续的语句才能访问该变量。访问变量包括向变量中写入数据，或读出其中的数据这两种操作。变量的**作用域**（scope）指的是 C++源程序中可以访问该变量的代码区域。C++语言按照定义位置将变量分为局部变量、全局变量和函数形参等三种类型，它们具有不同的作用域。所有变量只能在其作用域范围内访问。

**1）局部变量具有块作用域**

C++源程序中用一对大括号括起来的代码称为一个**代码块**。例如函数的函数体是一个代码块，一条复合语句也是一个代码块。**块作用域**（或称局部作用域，local scope）是从变量定义位置开始，一直到其所在代码块右大括号为止的区域。局部变量具有块作用域，只能被其块作用域内的语句访问。

**2）全局变量具有文件作用域**

**文件作用域**（或称全局作用域，global scope）是从变量定义位置开始，一直到其所在源程序文件结束为止的区域。全局变量具有文件作用域，可以被文件作用域内的所有函数访问。

**3）函数形参的作用域**

- 函数定义中的形参具有块作用域，这里的代码块指的是该函数的函数体。函数定义中的形参只能被本函数体内的语句访问。
- 函数声明中的形参不能也不需要被访问，其作用域可理解为空，称为**函数原型作用域**（function prototype scope）。声明函数时，其形参列表可以只声明数据类型，形参名可以省略。函数声明中形参名的作用仅仅是为了便于调用该函数的程序员理解参数的含义，没有其他语法作用。

例 5-11 给出一个演示不同类型变量及其作用域的 C++程序，其功能是计算 $x=y^2+z^2$。为了演示语法，程序被刻意划分成了 3 个函数（一个主函数和两个子函数），并分别定义了不同类型的变量。

**例 5-11　演示不同类型变量及其作用域的 C++程序**

```
#include <iostream>
using namespace std;

int fun1(int p1, int p2);          // fun1 函数声明：计算 p1 和 p2 的平方和。
                                   // 形参 p1 和 p2 具有函数原型作用域
```

```
6   int fun2(int p);     // fun2 函数声明：计算 p 的平方。形参 p 具有函数原型作用域
7
8   int x = 0;   // 全局变量 x 用来存储平方和
9
10  int main( )   // 主函数定义
11  {
12      cout << "Function main." ;   // 执行主函数时显示该信息
13
14      int y = 5, z = 10;   // 局部变量 y 和 z
15
16      x = fun1(y, z);   // 调用函数 fun1 计算 y 和 z 的平方和
17
18      cout << "计算结果为：" << x << endl;
19      return 0;
20  }
21
22  int fun1(int p1, int p2)   // fun1 函数定义：计算 p1 和 p2 的平方和
23  {
24      cout << "Function fun1." ;   // 执行函数 fun1 时显示该信息
25
26      int result;   // 局部变量 result
27      result = fun2( p1 );   // 计算 p1 的平方
28      result += fun2(p2);   // 计算 p2 的平方
29      return result;
30  }
31
32  int fun2(int p)   // fun2 函数定义：计算 p 的平方
33  {
34      cout << "Function fun2." ;   // 执行函数 fun2 时显示该信息
35
36      int result;   // 局部变量 result
37      result = p * p;   // 计算 p 的平方
38      return result;
39  }
```

全局变量 x 具有**文件作用域**

局部变量 y 和 z 具有**块作用域**

形参 p1 和 p2 具有**块作用域**

局部变量 result 具有**块作用域**

形参 p 具有**块作用域**

局部变量 result 具有**块作用域**

例 5-11 程序的说明如下：

（1）代码第 4 行。第 4 行函数 fun1 声明语句中的形参 p1 和 p2 具有函数原型作用域，即作用域为空，不能被访问。类似的还有第 6 行函数 fun2 声明语句中的形参 p 也是具有函数原型作用域。

（2）代码第 8 行。全局变量 x 具有文件作用域，即从第 8 行定义位置开始一直到第 39 行整个程序文件结束为止。该作用域内的所有函数都可以访问变量 x，例如主函数中代码第 16 行对 x 进行赋值，第 18 行显示 x 的值。在变量 x 的定义语句（第 8 行）之前访问 x 是错误的。

（3）代码第 14 行。局部变量 y 和 z 具有块作用域，即从第 14 行定义位置开始一直到第 20 行本代码块右大括号为止。该作用域内的所有语句都可以访问变量 y 和 z，例如代码第 16 行的函数调用语句会读取 y 和 z 的值，并将它们作为实参值传递给函数 fun1。在第 14 行~20 行之外的任何地方访问 y 或 z 都是错误的。

（4）代码第 22 行。函数 fun1 中定义的形参 p1 和 p2 具有块作用域，这里的代码块指的是该函数的函数体，即第 23 行~30 行之间的区域。fun1 函数体内的语句都可以访问 p1 和 p2，例如代码第 27、28 行的函数调用语句分别读出 p1 和 p2 中的数据（该数据是从主函数中传来的 y 和 z 的值），再将它们作为实参值继续传递给函数 fun2。形参只能被本函数访问，在第 23 行~30 行之外的地方访问 p1 或 p2 都是错误的。类似的还有第 32 行，函数 fun2 中定义的形参 p 也具有块作用域，只能被 fun2 函数体内的语句访问。

（5）代码第 26 行。result 是函数 fun1 中定义的局部变量，具有块作用域，即第 26 行~30 行之间的区域，只有这个区域内的语句才能访问 result。函数 fun2 中也定义了一个局部变量 result（第 36 行），其块作用域是第 36 行~39 行之间的区域。这两个 result 不是同一个变量。虽然名字相同，但它们分属于不同的函数。C++语言中，不同作用域的变量可以重名。

C++源程序中，变量遵循“先定义，后访问”的原则，函数则应遵循“先定义，后调用” 的原则。但函数也可以“先声明，后调用”，这样被调函数的定义代码可以放在调用该函数的语句后面。全局变量也可以“**先声明，后访问**”，即可以先访问，后定义，但需要在访问语句之前对该全局变量进行声明。

全局变量的文件作用域是从定义位置开始，一直到其所在源程序文件结束为止。可以使用 **extern** 关键字对全局变量进行声明，其作用是将该变量的作用域从定义位置往前延伸，请看下面的例子。

**程序 1：将全局变量定义在源程序的开头**

```
#include <iostream>
using namespace std;

int r;    // 将全局变量 r 定义在源程序的开头

int main( )
{
      cin >> r;
      cout << (3.14 * r * r);
      return 0;
}
```

**程序 2：将全局变量定义在源程序的末尾**

```
#include <iostream>
using namespace std;

extern int r;    // 声明后面定义的全局变量 r

int main( )
{
      cin >> r;
      cout << (3.14 * r * r);
      return 0;
}
int r;    // 将全局变量 r 定义在源程序的末尾
```

程序 1 将全局变量 r 定义在源程序文件的开头，其作用域一直到文件结束，因此主函数可以访问该变量。而程序 2 将全局变量 r 定义在源程序文件的末尾，主函数不在其作用域范围之内，不能访问。使用关键字 extern 对全局变量 r 进行声明，将其作用域从定义位置往前延伸到主函数定义代码之前，这样主函数就可以访问变量 r 了。

可以认为函数也是有作用域的，其作用域是从定义位置开始，一直到其所在源程序文件结束为止，即函数具有文件作用域。函数只能被其作用域内的函数调用，通过原型声明语句可以延伸函数的作用域。

需要注意的是，局部变量或函数形参不能通过声明语句延伸作用域。

### 2. 变量的重名规则

C++语言规定：同一作用域中的变量不能重名，不同作用域中的变量可以重名。例如，不同的函数可以定义重名的局部变量或形式参数，它们属于不同的块作用域。例 5-11 中，函数 fun1 和 fun2 都定义了一个名为 result 的局部变量。虽然名字相同，但它们不是同一个变量。例 5-12 给出另一个演示重名变量的 C++示例程序。

**例 5-12　演示重名变量的 C++示例程序**

```
1   #include <iostream>
2   using namespace std;
3
4   int x = 10;              // 定义 int 型全局变量 x，初始化为 10
5   double x = 1.5;          // 定义 double 型全局变量 x，初始化为 1.5。语法错误：不能重名
6
7   int main( )
8   {  // 函数体是一个语句块
9       cout << x << endl;          // 显示第 4 行变量 x 的值：10
10
11      int x = 20;                 // 定义 int 型局部变量 x，初始化为 20
12      double x = 2.5;   // 定义 double 型局部变量 x，初始化为 2.5。语法错误：不能重名
13
14      cout << x << endl;          // 显示第 11 行局部变量 x 的值：20
15
16      for (int n = 1; n <= 5; n++)
17      {  // 该复合语句也是一个语句块，包含在外层的函数体语句块中
18          int x = 30;             // 再定义 int 型局部变量 x，初始化为 30
19          double x = 3.5;   // 定义 double 型局部变量 x，初始化为 3.5。语法错误：不能重名
20
21          cout << x << endl;      // 显示第 18 行局部变量 x 的值：30
22      }
23
24      cout << x << endl;          // 显示第 11 行局部变量 x 的值：20
25      return 0;
26  }
```

例 5-12 程序的说明如下：

（1）代码第 5 行。所定义的新变量与第 4 行的变量 x 重名，它们属于同一文件作用域，不能重名。

（2）代码第 11 行。所定义的新变量虽然与第 4 行的变量 x 重名，但它们属于不同作用域，可以重名。第 4 行定义的全局变量 x 具有文件作用域，该作用域包含第 11 行局部变量 x 的函数体块作用域。具有包含关系的作用域不是同一作用域，不同作用域中的变量可以重名。

（3）代码第 12 行。所定义的新变量与第 11 行的变量 x 重名，它们属于同一块作用域，不能重名。

（4）代码第 18 行。所定义的新变量虽然与第 4、11 行的变量 x 都重名，但它们分属于不同作用域。第 18 行的变量 x 是在 for 循环体中定义的局部变量，具有复合语句的块作用用

域。该作用域被包含在第 11 行变量 x 的函数体块作用域中，两者之间存在包含关系。第 11 行变量 x 的函数体块作用域又包含在第 4 行变量 x 的文件作用域中。上述 3 个变量的作用域具有逐层包含关系，但不是同一作用域，可以重名。

（5）代码第 19 行。所定义的新变量与第 18 行的变量 x 重名，它们属于同一块作用域，不能重名。

一个 C++源程序文件可以包含多个函数，函数的函数体中又可以包含复合语句，复合语句还可以再包含下层的复合语句。文件作用域是最外层的作用域，它包含函数体块作用域。函数体块作用域可以包含复合语句的块作用域，外层复合语句的块作用域还可以包含更内层复合语句的块作用域。具有包含关系的作用域不是同一作用域，不同作用域中可以定义重名的变量。

不同函数的函数体块作用域之间是并列关系，不是同一作用域。因此不同函数可以定义重名的形参或局部变量。

为了便于实际应用，我们对变量的重名规则做如下归纳总结：

（1）局部变量、函数形参可以和全局变量重名。

（2）不同的函数可以定义重名的形参或局部变量。

（3）在复合语句内部可以定义与其外部重名的变量。

（4）函数中不能定义与形参重名的局部变量，除非将该变量定义在某个复合语句里面。

### 3. 访问重名变量的局部优先原则

C++语言中的所有变量都只能在其作用域范围内访问。换句话说，只要在变量作用域范围内的语句就都可以访问该变量。假设有两个重名变量 x，其作用域具有包含关系，其中一个 x 定义在外层，另一个 x 定义在内层。内层作用域中的语句可以同时访问这两个重名变量，此时通过变量名 x 访问会出现二义性，即访问的到底是哪一个 x 呢？C++规定：内层作用域中的语句访问重名变量时，优先访问内层定义的变量，这就是访问重名变量时的**局部优先原则**。

例 5-12 中使用 cout 语句访问重名变量 x，它们到底会显示哪一个 x 的值呢？程序执行时，计算机会顺序执行主函数中的代码，请看下面的代码执行说明。

（1）代码第 9 行。此时只定义了全局变量 x，没有其他重名变量，因此 cout 语句将显示该全局变量的值 10。

（2）代码第 14 行。此时又定义了一个局部变量 x，有两个重名的变量 x。根据局部优先的原则，cout 语句将显示局部变量 x 的值 20。

（3）代码第 21 行。在复合语句中又定义了一个局部变量 x，此时共有 3 个重名的变量 x。根据局部优先的原则，cout 语句将显示复合语句中变量 x 的值 30。

（4）代码第 24 行。此时复合语句已经执行结束，其中的局部变量 x 即被删除，还剩下两个重名的变量 x。根据局部优先的原则，cout 语句与将和第 14 行一样，显示第 11 行局部变量 x 的值 20。

有了变量重名和局部优先原则，程序员在函数中定义局部变量时，只要单纯考虑自己的变量之间不要重名就可以了，与其他函数无关。这样程序员就可以各自编写函数代码，互不干扰。

### 本节习题

1. 下列哪种变量属于局部变量？（　　）
   A. 定义在函数体中的变量　　B. 定义在函数外部的变量
   C. 定义在函数头中的变量　　D. 定义在函数原型声明中的变量
2. 函数调用语句中的实参是按什么原则一一赋值给被调函数形参的？（　　）
   A. 按位置顺序　　B. 按位置顺序的相反顺序
   C. 按参数名相同的原则　　D. 按数据类型相同的原则
3. 下列关于全局变量的叙述，错误的是（　　）。
   A. 全局变量定义在函数的外部
   B. 全局变量只能被其定义语句后面的函数访问
   C. 全局变量不属于任何函数
   D. 多个函数可通过全局变量共享数据
4. 下列关于变量作用域的叙述，错误的是（　　）。
   A. 变量只能在其作用域范围内访问　　B. 局部变量具有文件作用域
   C. 局部变量具有块作用域　　D. 全局变量具有文件作用域
5. 下列关于变量重名的叙述，错误的是（　　）。
   A. 同一作用域中的变量不能重名
   B. 不同作用域中的变量可以重名
   C. 在内层作用域中访问重名变量遵循局部优先原则
   D. 在内层作用域中访问重名变量遵循外部优先原则
6. 计算机执行下列 C++语句：

```
int x = 1;
{
        cout << x;
        int x = 2;
        {
                int x = 3;      cout << x;
        }
        cout << x;
}
```

   执行结束后，显示器将显示（　　）。
   A. 123　　B. 321　　C. 1, 3, 2　　D. 132

## 5.4 程序代码和变量的存储原理

C++语言采用编译执行的方式。程序员编写的 C++源程序需经过编译、连接，翻译成等效的目标程序（即机器语言程序），计算机执行目标程序。平时，目标程序是以可

执行文件的形式保存在外存（比如硬盘）上的，执行时被读入内存，在内存中建立一个**程序副本**。

操作系统可以认为是计算机开机后执行的第一个程序。计算机运行期间，操作系统的程序副本一直停留在内存中。当用户执行某个应用程序时，操作系统将该程序从外存读入内存。多任务操作系统可同时运行多个程序，每个程序都有自己的内存副本。当程序执行结束退出时，程序副本所占用的内存被释放。计算机运行期间，操作系统负责管理、分配系统中未被占用的空闲内存。系统空闲内存被称为**堆**（heap）。

## 5.4.1　程序副本与变量

本节以例 5-11 所示的 C++程序为例，具体讲解程序执行时程序副本和变量在内存中的存储原理。

### 1. 加载程序

当执行例 5-11 所示的 C++程序时，操作系统将其可执行文件读入内存，建立一个程序副本，这个过程称为**加载**程序。图 5-6(a)给出加载例 5-11 程序后的内存示意图，阴影部分表示该程序的副本。程序副本中包括代码区、常量区、静态存储区和自动存储区 4 个部分。

- **代码区**：存放的是程序中各函数所对应的机器语言代码，例如主函数 main 和两个子函数 fun1、fun2。
- **常量区**：存放的是程序中的某些常量，例如本例中的 3 个字符串常量。这些常量在程序加载后即占用内存空间。
- **静态存储区**：存放的是程序中定义的全局变量，例如变量 x。全局变量在程序加载后即分配内存空间。
- **自动存储区**：此处将存放程序中定义的局部变量和函数形参。自动存储区又称为**栈**（stack），是由编译器为程序预留的存储空间，有大小限制（可在编译器中设置）。程序加载时自动存储区还是空的，开始执行后将从中为局部变量和函数形参分配内存。

程序加载后立即为程序中的全局变量分配内存。全局变量将一直占用所分配的内存，直到程序执行结束退出时才被释放，这种内存分配方法称为**静态分配**。静态分配的全局变量被存放在静态存储区。

在函数或复合语句中定义的局部变量，它们在计算机执行到其定义语句时才分配内存，到其所在代码块执行结束即被释放，这种内存分配方法称为**自动分配**。函数定义中的形参也是自动分配内存的，当执行到该函数的调用语句时为形参分配内存并在其中写入传递来的实参值，然后执行函数体，函数体执行结束后立即释放形参的内存。自动分配的局部变量和形参被存放在自动存储区。

程序从加载到执行结束退出，这个时间段是一个程序在内存中的**生存期**（storage duration）。变量从内存分配到释放，这个时间段就是一个变量在内存中的生存期。

| 内容 | 区域 |
|---|---|
| 操作系统<br>及已运行的应用程序 | |
| int main( )<br>{ … } | 代码区 |
| int fun1(int p1, int p2)<br>{ … } | |
| int fun2(int p)<br>{ … } | |
| "Function main. "<br>"Function fun1. "<br>"Function fun2. " | 常量区 |
| x: 0 | 静态存储区 |
| 栈剩余内存 | 自动存储区(栈) |
| 系统空闲内存 | 堆 |

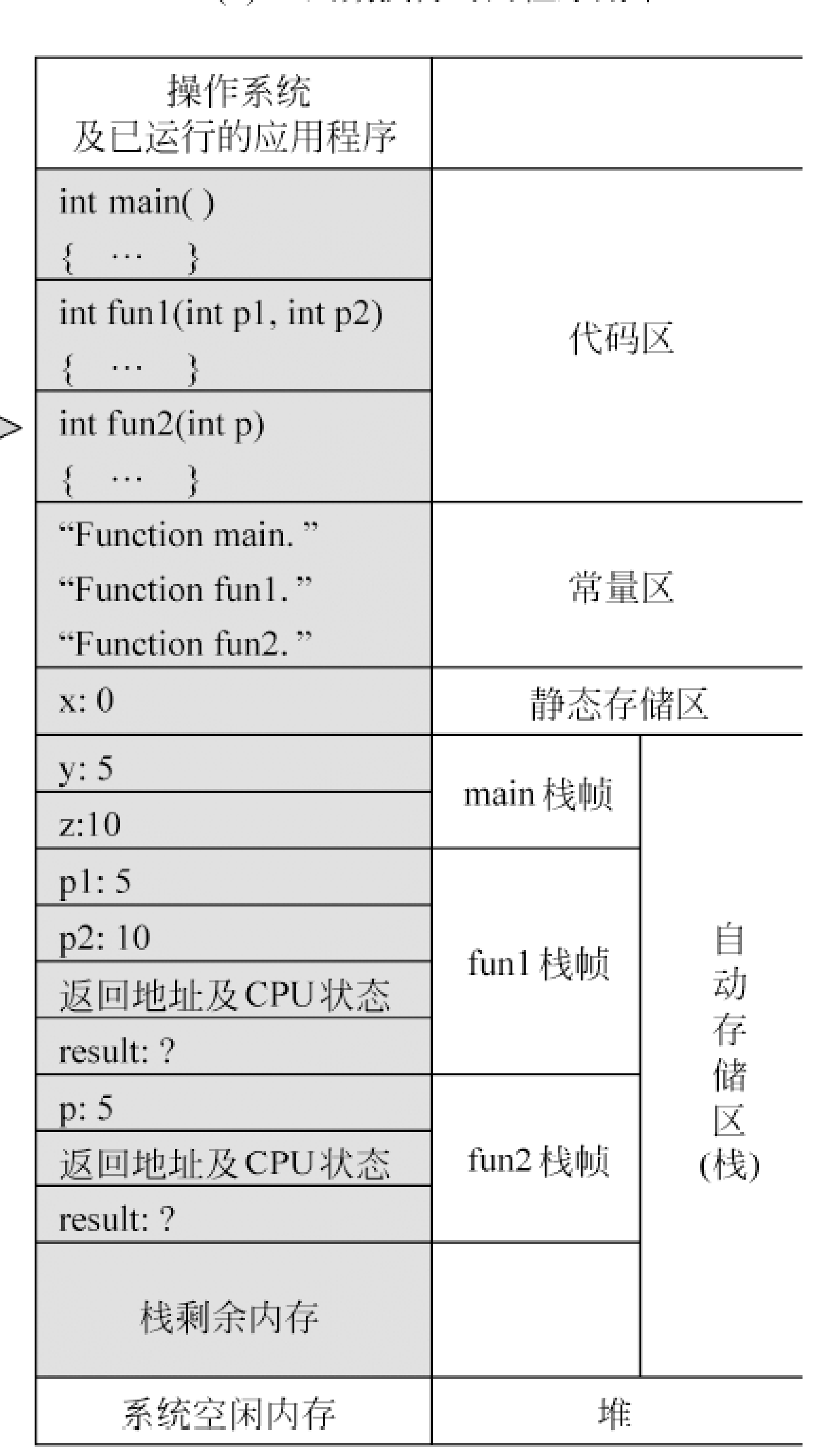
(a) 加载后的程序副本

| 内容 | 区域 | |
|---|---|---|
| 操作系统<br>及已运行的应用程序 | | |
| int main( )<br>{ … } | 代码区 | |
| int fun1(int p1, int p2)<br>{ … } | | |
| int fun2(int p)<br>{ … } | | |
| "Function main. "<br>"Function fun1. "<br>"Function fun2. " | 常量区 | |
| x: 0 | 静态存储区 | |
| y: 5<br>z:10 | main 栈帧 | 自动存储区(栈) |
| 栈剩余内存 | | |
| 系统空闲内存 | 堆 | |

(b) 主函数执行时的程序副本

| 内容 | 区域 | |
|---|---|---|
| 操作系统<br>及已运行的应用程序 | | |
| int main( )<br>{ … } | 代码区 | |
| int fun1(int p1, int p2)<br>{ … } | | |
| int fun2(int p)<br>{ … } | | |
| "Function main. "<br>"Function fun1. "<br>"Function fun2. " | 常量区 | |
| x: 0 | 静态存储区 | |
| y: 5<br>z:10 | main 栈帧 | 自动存储区(栈) |
| p1: 5<br>p2: 10<br>返回地址及CPU状态<br>result: ? | fun1栈帧 | |
| 栈剩余内存 | | |
| 系统空闲内存 | 堆 | |

(c) 调用函数fun1时的程序副本




(d) 调用函数fun2时的程序副本

图 5-6　执行例 5-11 程序时的内存示意图

### 2. 执行主函数

程序加载后，CPU 开始执行程序，从主函数的第一条语句开始执行。当执行到定义变量语句：

```
int y = 5, z = 10;
```

时，计算机为变量 y 和 z 分配内存，如图 5-6(b)所示。

计算机在执行函数时才为其形参及函数体中定义的局部变量分配内存，这些内存合在一起被称为该函数执行时的**栈帧**（stack frame），因为它们是从程序副本预留的内存栈中分配来的。栈帧中的形参或局部变量只能被本函数中的语句直接访问。

本例中，主函数没有形参，其函数体定义了两个局部变量 y 和 z，因此图 5-6(b)中的主函数 main 栈帧将只包括这两个变量。图 5-6(b)中的箭头表示当前 CPU 正在执行主函数 main。

### 3. 主函数调用子函数 fun1

计算机执行函数调用语句的内部细节如下。

（1）首先为被调函数的形参分配内存，然后计算调用语句中的实参表达式，将表达式结果按位置顺序一一赋值给对应的形参。

（2）被调函数执行结束返回主调函数时，计算机需要知道调用前已经执行到哪儿了，下一步该执行哪条指令，并且还要将 CPU 恢复到调用前的状态。因此在跳转去执行被调函数前，计算机会自动保存好返回地址以及当前 CPU 状态，这些信息被临时保存在被调函数的栈帧中。返回地址是返回主调函数时 CPU 将执行的下一条指令地址。保存当前 CPU 状态，被称为**保存现场**。

（3）保存好返回地址及 CPU 状态后，计算机才正式开始执行被调函数的函数体，并在执行结束后返回主调函数。返回前，首先将返回值传回主调函数，然后将 CPU 恢复到调用前的状态，即**恢复现场**。最后按照返回地址继续执行主调函数中剩余的指令，调用结束。

本例中，当执行到主函数中调用子函数 fun1 的语句：

```
x = fun1(y, z);
```

时，计算机将暂停主函数的执行，跳转去执行子函数 fun1。跳转前首先为函数 fun1 定义的形参 p1 和 p2 分配内存，并将调用语句中的实参值（即变量 y 和 z 的值，分别为 5 和 10）按位置顺序一一赋值给对应的形参 p1 和 p2，保存好返回地址和 CPU 状态，然后开始执行子函数 fun1 的函数体。当执行到函数 fun1 中的定义变量语句：

```
int result;
```

时，计算机为变量 result 分配内存。图 5-6(c)显示了此时的内存示意图，其中新增的 fun1 栈帧包含 p1、p2、result 这三个变量，以及返回地址和 CPU 状态信息。

### 4. 子函数 fun1 嵌套调用另一个子函数 fun2

当执行到函数 fun1 中调用另一个子函数 fun2 的语句：

```
result = fun2(p1);
```

时，计算机将暂停 fun1 的执行，跳转去执行函数 fun2。跳转前首先为函数 fun2 定义的形参 p 分配内存，并将调用语句中的实参值（即 p1 的值，为 5）赋值给对应的形参 p，保存好返回地址和 CPU 状态，然后开始执行 fun2 的函数体。当执行到其中的定义变量语句：

```
int result;
```

时，计算机为变量 result 分配内存，如图 5-6(d)所示，其中新增的 fun2 栈帧包含 p 和 result 这两个变量。可以看到此时内存中有两个 result 变量，但它们不在同一栈帧，有各自的内存单元，分别属于不同的函数（即 fun1 和 fun2）。当前 fun2 处于执行状态。此时访问变量 result 所访问的是 fun2 栈帧中 result 的内存单元。

可以看出，每增加一级函数调用，内存中的栈帧就增加一个。

#### 5. 函数 fun2 执行结束返回主调函数 fun1

当执行到函数 fun2 中的 return 语句：

```
return result;
```

时，首先将 result 中的值返回给主调函数 fun1（本次返回值为 25），然后退出函数 fun2 的执行。计算机在退出函数 fun2 时会释放其栈帧，其中形参 p 和变量 result 的生存期结束，所保存的数据也被丢弃，内存回到图 5-6(c)的状态。此时 fun1 栈帧中的变量 result 将被赋值为 25（即 fun2 的返回值）。将 CPU 恢复到调用前的状态，返回主调函数 fun1，按照返回地址继续执行其中剩余的指令。

可以看出，计算机会在函数执行结束退出时释放其栈帧，其中所有形参和局部变量的生存期结束，所保存的数据也被丢弃。每退出一级函数调用，内存中的栈帧就减少一个。

#### 6. 函数 fun1 第 2 次调用 fun2

当执行到函数 fun1 中第 2 条调用函数 fun2 的语句：

```
result += fun2(p2);
```

时，计算机再次暂停 fun1 的执行，跳转去执行函数 fun2。调用过程和上述第 4 和第 5 步所描述的过程完全一样，再次为 fun2 中的形参和局部变量分配内存，函数执行结束返回时释放。所不同的是，第 2 次调用所传递的实参值是 p2 的值 10，传递回来的返回值是 100。调用结束后，fun1 栈帧中变量 result 被累加到 125。

#### 7. 函数 fun1 执行结束返回主函数

当执行到函数 fun1 中的 return 语句：

```
return result;
```

时，计算机将 fun1 栈帧中 result 的值 125 返回给主函数，然后退出函数 fun1 的执行。计算机在退出函数 fun1 时会释放其栈帧，其中形参 p1、p2 和变量 result 的生存期结束，所保存的数据也被丢弃，内存回到图 5-6(b)的状态。此时全局变量 x 将被赋值为 125（即 fun1 的返回值）。将 CPU 恢复到调用前的状态，返回主函数 main，按照返回地址继续执行其中

剩余的指令。

**8. 主函数执行结束，程序退出**

当执行到主函数的最后一条指令：

```
return 0;
```

时，计算机将结束程序的执行并退出。退出时会释放整个程序副本所占用的内存空间，此时全局变量 x 的生存期也结束了。所释放的内存被作为空闲内存交还给操作系统管理。

变量从内存分配到释放，这个时间段是一个变量在内存中的生存期。从上述 C++程序的执行过程可以清楚地看到，不同变量具有不同的生存期。

全局变量从程序一加载即被分配内存，直到程序执行结束退出时才被释放。全局变量的生存期和程序生存期一样长。

在函数或复合语句中定义的局部变量，它们是在计算机执行到其定义语句时才分配内存，到其所在代码块执行结束即被释放。一个函数或一条复合语句可能被执行多次。随着函数或复合语句的执行，其中所定义的局部变量将不断重复内存的“分配－释放”过程。这个过程是自动完成的，不需要程序员干预。函数中的形参也会随着函数的“调用－退出”而不断重复内存的“分配－释放”过程。局部变量和形参的生存期比程序的生存期短。

本节最后，我们再简单回顾一下计算机执行函数调用语句的具体过程。

（1）计算机执行到函数调用语句时将暂停主调函数的执行，跳转去执行被调函数。

（2）跳转前首先为被调函数定义的形参分配好内存，然后计算调用语句中的实参表达式，并将表达式结果按位置顺序一一赋值给对应的形参，这就是形实结合。

（3）保存好返回地址和当前 CPU 状态，即保存调用前的现场。

（4）跳转去执行被调函数的函数体。

（5）执行完被调函数的函数体或执行到其中的 return 语句时，退出被调函数的执行。如果有返回值，将返回值传回主调函数。

（6）恢复主调函数调用被调函数之前的现场，调用结束。计算机按照返回地址继续执行主调函数中剩余的指令。

## 5.4.2　动态分配的内存

第 4 章 4.2.2 节曾介绍过一种动态内存分配方法。程序员可以根据实际需要，在程序中使用 new 运算符来分配内存，使用完之后使用 delete 运算符将其释放。这种**动态分配**方法可以让程序员更主动、更直接地管理内存，根据需要分配尽可能少的内存，同时尽早释放以减少内存的占用时间。

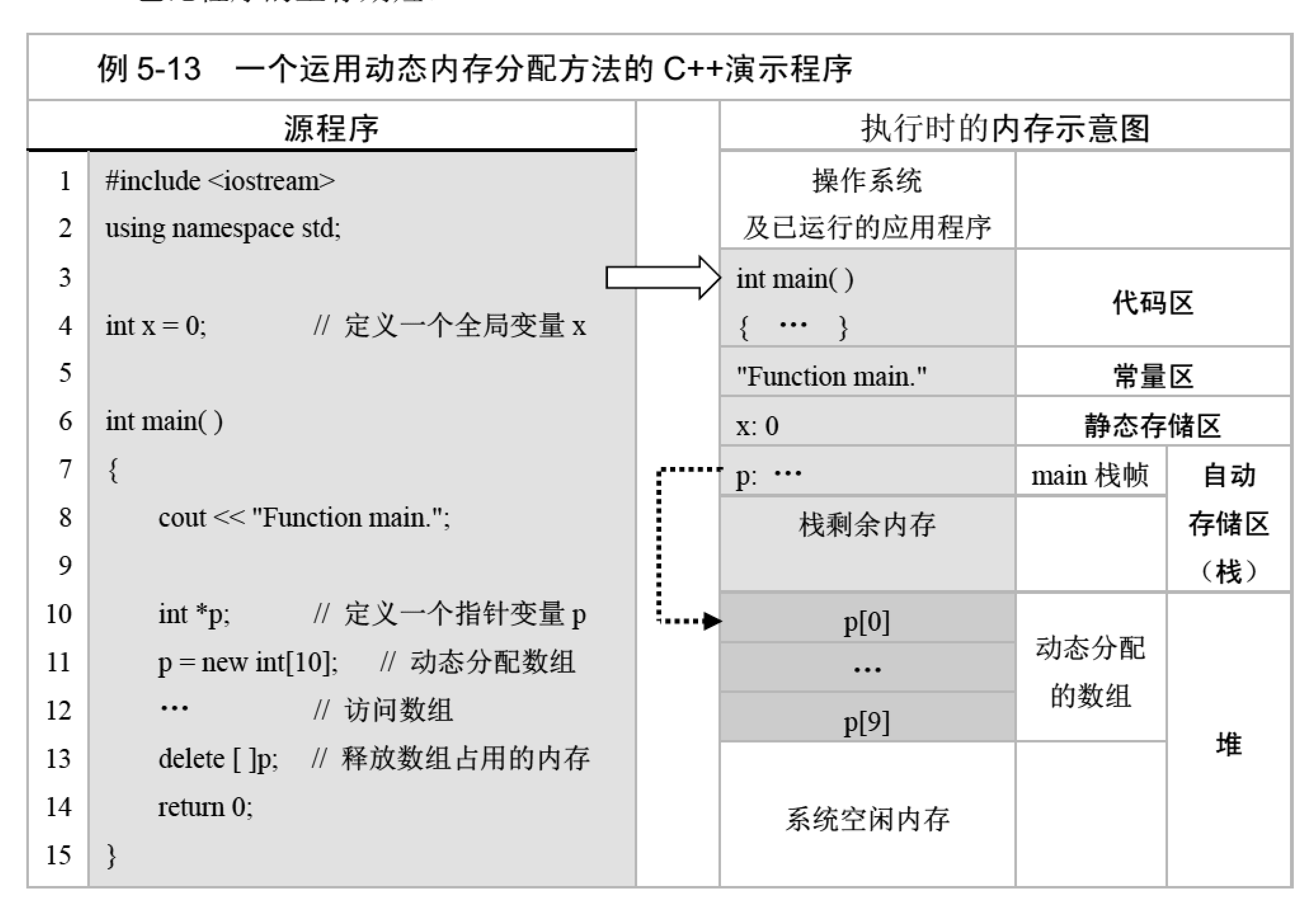
动态分配的内存是从系统空闲内存（即堆中）分配来的，需要程序员自己释放，否则即使程序结束退出，这个内存也一直被占用，无法被操作系统回收再利用，这称为**内存泄漏**。

内存的动态分配、访问及释放都是通过内存地址实现的，因此需要另外定义一个指针变量来保存地址。例 5-13 是一个运用动态内存分配方法的 C++演示程序，右侧给出了程序

执行时的内存示意图。

例 5-13 中的 C++程序使用 new 运算符动态分配一个含有 10 个元素的 int 型数组，计算机从堆中为其分配内存单元，并返回该内存单元的首地址。将返回的首地址保存到预先定义好的指针变量 p 中。从例 5-13 右侧的内存示意图可以看出以下几点：

- **全局变量**（例如 x）是静态分配的，存放在程序副本中的静态存储区。其生存期与程序一样长。
- **局部变量**（例如指针变量 p）是自动分配的，存放在程序副本的栈中。其生存期由其定义位置和所在代码块的结束语句决定，但肯定比程序的生存期短。
- **动态分配**的内存则存放在系统的堆中。其生存期由 new 和 delete 运算符的位置决定，也比程序的生存期短。




## 5.4.3　函数指针

可以通过内存地址访问变量，也可以通过内存地址调用函数。计算机程序在执行时被读入内存，在内存中建立一个程序副本，其中包括各函数的代码。也就是说，执行时程序中各函数的代码是存放在内存中的。调用函数一般是通过函数名来调用，也可以通过函数代码的首地址来调用。通过地址调用函数需分为 3 步，依次是定义函数型指针变量，将函数首地址赋值给该指针变量，再通过指针变量间接调用函数。

### 1. 定义函数型指针变量

C++语法：定义函数型指针变量

**函数类型 (*指针变量名)(形式参数列表)；**

语法说明：

- 函数型指针变量需要与函数匹配。"**匹配**"的含义是在定义函数型指针变量时，除了在变量名前加指针说明符"*"之外，同时还需要指定函数的形式参数和返回值类型。
- **形式参数列表**和**函数类型**分别指定了函数的形参和返回值类型，其中的形参名可以省略。指针变量将只能指向具有指定形参和返回值类型的函数。
- **指针变量名**需符合标识符的命名规则。

假设有如下的示例函数：

```
double fun1(double x, int y)   {   return (x+y);   }   // 该函数实现简单的加法功能
```

可以定义一个与上述函数匹配的指针变量 p：

```
double (*p)(double, int);                     // 定义函数型指针变量 p
```

### 2. 将函数首地址赋值给该指针变量

可以使用取地址运算符"&"来获取执行时程序中变量的内存地址。该如何获取函数的内存地址呢？和数组名一样，函数名也可理解为是一个表示函数首地址的符号常量。例如：

```
p = fun1;                     // 将函数 fun1 的内存首地址赋值给指针变量 p
```

注意：*函数和指针变量之间需互相匹配才能赋值。*

### 3. 通过指针变量间接调用函数

通过指针变量间接调用函数的语法形式是：

**(*函数型指针变量名)(实际参数列表)**

其中"*"是指针运算符，调用时需按照函数的要求给出具体的实参值。例如：

```
cout << (*p)(3.5, 2);          // 间接调用函数 fun1，实现简单的加法。显示结果：5.5
```

通过指针变量间接调用函数的语法形式也可以是：

**函数型指针变量名（实际参数列表）**

这种语法形式是将指针变量名直接当做函数名来使用，调用形式更加简洁。例如：

```
cout << p(3.5, 2);             // 间接调用函数 fun1，形式更加简洁。显示结果：5.5
```

通过上述例子可以看出，函数名实际上可以理解成是指向内存中函数代码的指针，只不过它是一种指针常量，固定指向某个函数。

而一个函数型指针变量可以指向多个不同的函数。这些函数都需要与指针变量匹配，即它们具有相同的形参和返回值类型。例 5-14 给出一个应用函数型指针变量的 C++程序。

**例 5-14　应用函数型指针变量的 C++程序**

```
1   #include <iostream>
2   using namespace std;
3
4   // 下列 4 个函数具有相同的形参和函数类型
5   double fun1(double x, int y)          // 函数 fun1 实现简单的加法功能
6   {     return (x+y);     }
7   double fun2(double x, int y)          // 函数 fun2 实现简单的减法功能
8   {     return (x–y);     }
9   double fun3(double x, int y)          // 函数 fun3 实现简单的乘法功能
10  {     return (x*y);     }
11  double fun4(double x, int y)          // 函数 fun4 实现简单的除法功能
12  {     return (x/y);     }
13
14  int main( )  // 主函数将通过指针变量间接调用上述 4 个函数
15  {
16      double (*p)(double, int);          // 定义一个函数型指针变量 p
17      p = fun1;   cout << (*p)(3.5, 2) << endl;      // 间接调用 fun1 实现加法，显示结果：5.5
18      p = fun2;   cout << (*p)(3.5, 2) << endl;      // 间接调用 fun2 实现减法，显示结果：1.5
19      p = fun3;   cout << p(3.5, 2) << endl;         // 间接调用 fun3 实现乘法，显示结果：7.0
20      p = fun4;   cout << p(3.5, 2) << endl;         // 间接调用 fun4 实现除法，显示结果：1.75
21      return 0;
22  }
```

例 5-14 中，代码第 17、18 行使用“(*p)”的形式来间接调用函数，而第 19、20 行则将指针变量名 p 直接当做函数名使用。第二种调用形式更加简洁。

## 本节习题

1. 程序执行时，其内存副本不包括下列哪一项内容？（　　）
   A. 代码区　　B. 函数原型声明区
   C. 静态存储区　　D. 自动存储区
2. 下列关于变量内存分配的叙述，错误的是（　　）。
   A. 局部变量是自动分配的　　B. 局部变量的内存生存期与程序一样长
   C. 全局变量是静态分配的　　D. 全局变量的内存生存期与程序一样长
3. 下列关于变量内存分配的叙述，错误的是（　　）。
   A. 局部变量存储在静态存储区　　B. 局部变量存储在自动存储区
   C. 全局变量存储在静态存储区　　D. 动态分配是从系统堆中分配内存的
4. 通过指针变量不能实现下列哪种功能？（　　）
   A. 读出所指向内存变量中的数据　　B. 向所指向的内存变量中写入数据
   C. 调用所指向内存中存放的函数　　D. 访问内存中的任意内存单元

5. 函数 fun 中定义了一个局部变量 x：

```
void fun( )
{
    int   x;
     …
}
```

假设程序执行过程中，函数 fun 被调用了 3 次，则变量 x 经历了几次内存分配－释放的过程？（　　）

A. 0　　　　B. 1　　　　C. 2　　　　D. 3

# 5.5　函数间参数传递的三种方式

采用数据分散管理策略时，数据分散在各个函数中管理。函数各自定义局部变量保存数据，其他函数不能直接访问。调用函数时，主调函数和被调函数之间需要通过形实结合来传递数据，将保存在主调函数里的原始数据以实参的形式传递给被调函数的形参，这称为函数间的**参数传递**。C++语言中，参数传递有三种方式，分别是值传递、引用传递和指针传递。本节通过一个程序实例来具体讲解这三种参数传递方式，程序实例的描述如下：

定义两个变量“int x = 5, y = 10;”，编写一个函数 swap 来交换这两个变量的数值。交换后，x 的值应为 10，y 的值应为 5。

## 5.5.1　值传递

**值传递**是将主调函数中实参的数值传递给被调函数的形参。形参单独分配内存，另外保存一份实参值的副本，被调函数访问的是形参中的副本。实际上到目前为止，本章所有示例程序的参数传递一直都是值传递方式。例 5-15 希望继续使用值传递方式来实现交换变量 x 和 y 数值的功能。

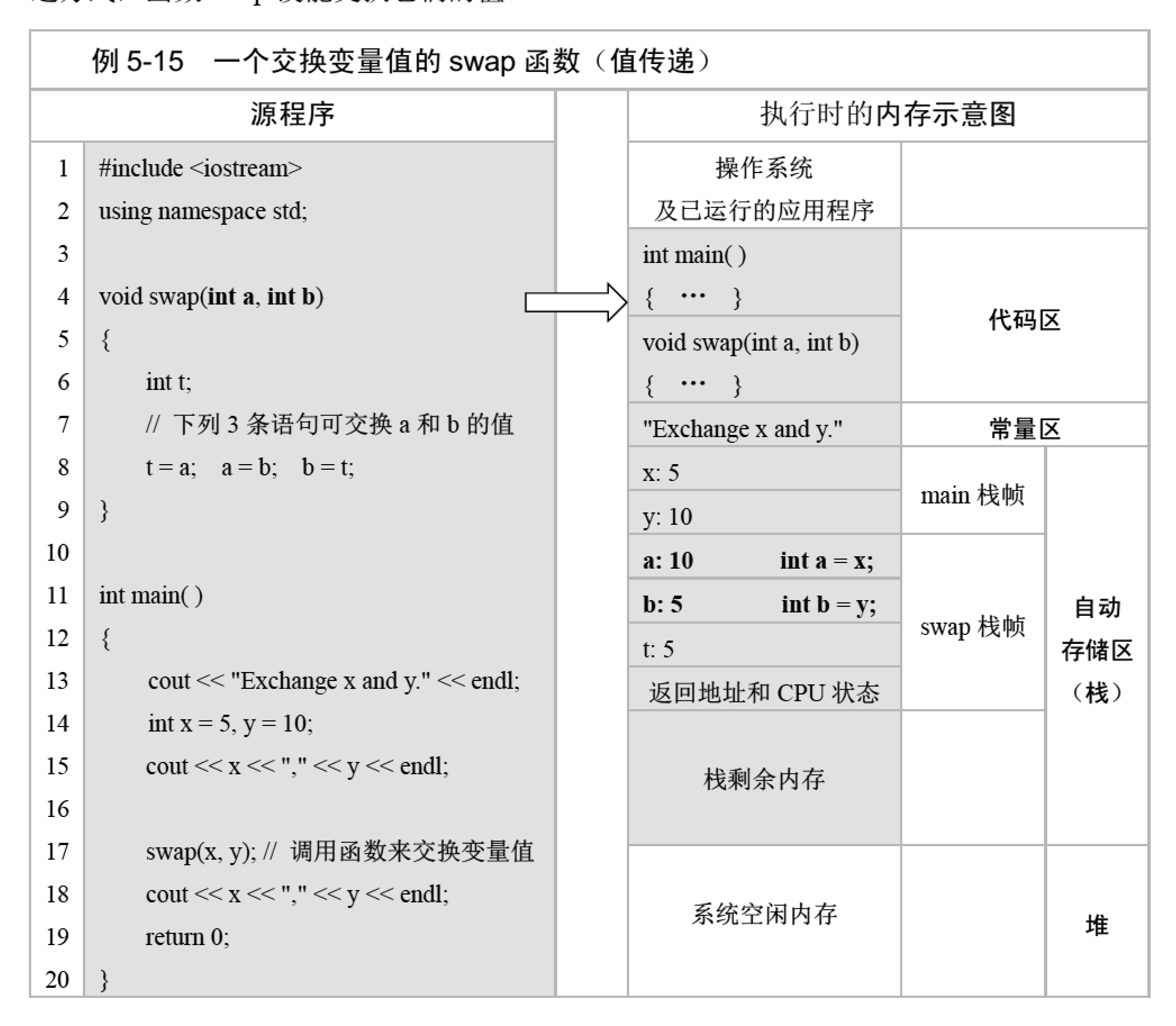
例 5-15 中，主函数希望通过调用函数 swap 来交换变量 x 和 y 的值，调用所传递的实参就是 x 和 y（第 17 行）。函数 swap 定义两个 int 型形参 a、b 来接收主函数中变量 x、y 的值（第 4 行）。主函数调用 swap 函数，值传递的参数传递过程相当于执行了下列语句：

```
int a = x;                    // 定义形参 a 并初始化为实参 x 的值
int b = y;                    // 定义形参 b 并初始化为实参 y 的值
```

这样函数 swap 中的形参 a、b 就分别保存了一份主函数中实参 x、y 的数据副本。函数 swap 再通过 3 条赋值语句来交换形参 a 和 b 的值（第 8 行），交换结束后返回主函数。主函数在函数调用语句的前后分别用一条 cout 指令来显示 x、y 的值（第 15、18 行），以验证 swap 函数是否实现了变量值的交换。

例 5-15 的右侧给出了函数 swap 刚刚执行结束，准备返回主函数之前那个时刻的内存示意图。可以看出，形实结合后实参 x、y 的值确实传给了形参 a、b，然后 a 和 b 的值被交换了，但 x、y 的值没有跟着改变。返回主函数后，swap 栈帧被释放，形参 a、b 的生存

期结束。主函数在函数调用语句之后再次显示 x 和 y 的值，它们依然是 5 和 10。采用值传递方式，函数 swap 没能交换它们的值。


例 5-15　一个交换变量值的 swap 函数（值传递）

源程序

```
#include <iostream>
using namespace std;

void swap(int a, int b)
{
      int t;
      // 下列 3 条语句可交换 a 和 b 的值
      t = a;   a = b;   b = t;
}

int main( )
{
      cout << "Exchange x and y." << endl;
      int x = 5, y = 10;
      cout << x << "," << y << endl;

      swap(x, y); // 调用函数来交换变量值
      cout << x << "," << y << endl;
      return 0;
}
```



值传递具有如下特点：

（1）值传递只是将主调函数中实参的值传递给被调函数的形参，通常用于将主调函数中的原始数据传递给被调函数。被调函数修改形参中的数据，修改的只是数据副本，不会影响主调函数中实参的数据。

（2）值传递只能将数据从主调函数传给被调函数，这是一种单向数据传递机制。

（3）值传递时，实参可以是常量、变量或表达式。

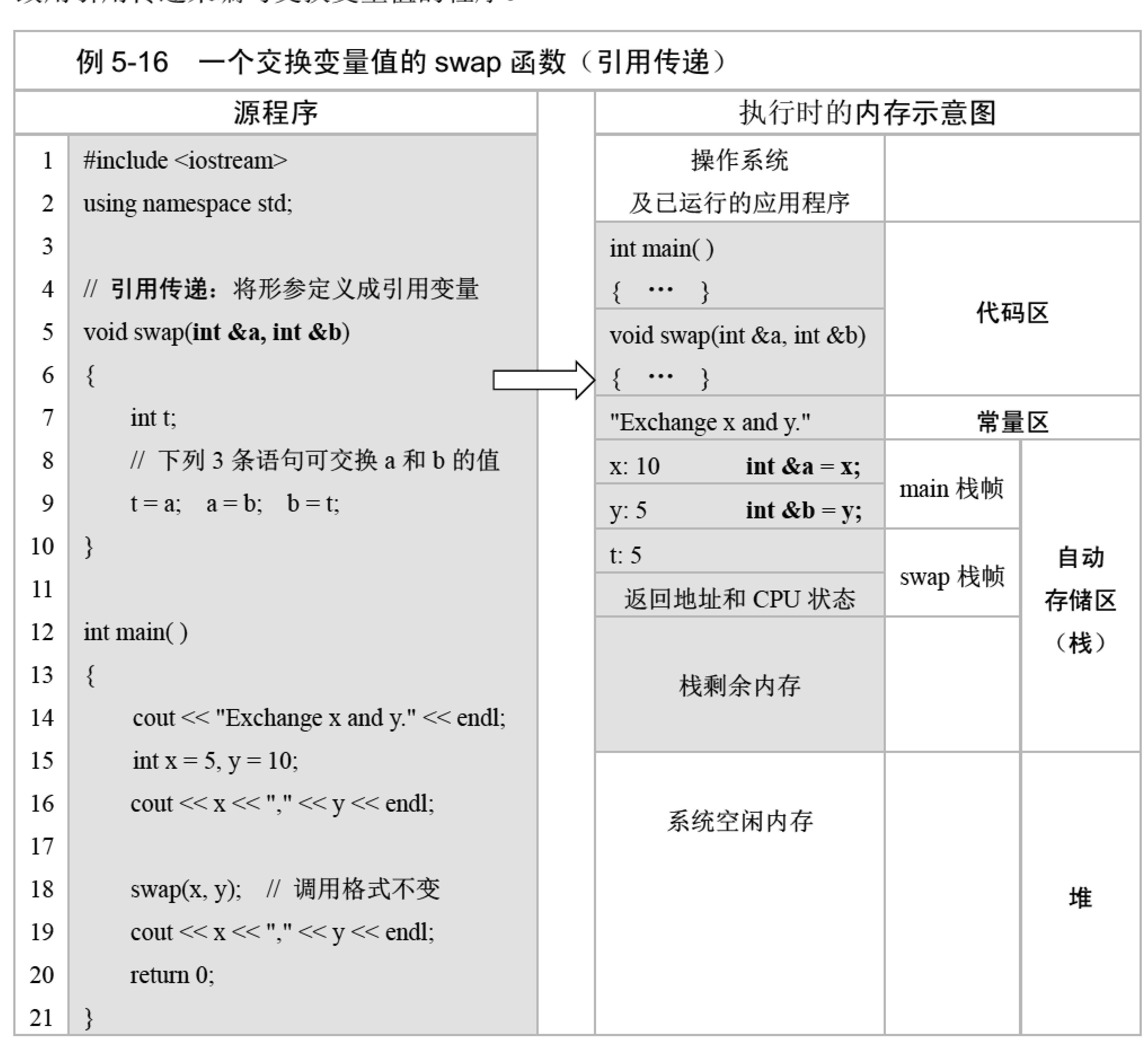
（4）值传递的好处是，如果编写被调函数的程序员错误修改了形参，这不会影响到主调函数中的实参。函数之间不会互相干扰。

## 5.5.2　引用传递

C++语言中，访问变量的内存单元有三种方式，分别是变量名、引用和指针。通过变量名访问称为直接访问，通过变量的引用（即别名）或指针（即内存地址）进行访问称为

间接访问。函数中定义的局部变量不能被其他函数直接访问，但能够被间接访问。

**引用传递**就是传递实参变量的引用，被调函数通过该引用间接访问主调函数中的变量，从而达到传递数据的目的。采用引用传递时，被调函数的形参需定义成引用变量。例 5-16 改用引用传递来编写交换变量值的程序。



例 5-16　一个交换变量值的 swap 函数（引用传递）

源程序

```
1   #include <iostream>
2   using namespace std;
3
4   // 引用传递：将形参定义成引用变量
5   void swap(int &a, int &b)
6   {
7       int t;
8       // 下列 3 条语句可交换 a 和 b 的值
9       t = a;   a = b;   b = t;
10  }
11
12  int main( )
13  {
14      cout << "Exchange x and y." << endl;
15      int x = 5, y = 10;
16      cout << x << "," << y << endl;
17
18      swap(x, y);   // 调用格式不变
19      cout << x << "," << y << endl;
20      return 0;
21  }
```



例 5-16 中，函数 swap 定义两个 int 型引用形参 a、b（第 5 行），主函数调用该函数的格式不变（第 18 行）。主函数调用 swap 函数，引用传递的参数传递过程相当于执行了下列语句：

```
int &a = x;        // a 是变量 x 的别名，访问 a 就是访问 x
int &b = y;        // b 是变量 y 的别名，访问 b 就是访问 y
```

函数 swap 交换 a、b 的值（第 9 行），实际上就是交换了主函数中变量 x、y 的值。

例 5-16 的右侧给出了函数 swap 刚刚执行结束，准备返回主函数之前那个时刻的内存示意图。可以看出，形参 a、b 没有分配内存，它们分别是实参 x、y 的别名。主函数在函数调用语句之后再次显示 x 和 y 的值，将显示 10 和 5。采用引用传递方式，函数 swap 成功交换了变量 x 和 y 的值。

引用传递具有如下特点：

（1）引用传递将被调函数的形参定义成主调函数中实参变量的引用，被调函数通过该引用间接访问主调函数中的变量。

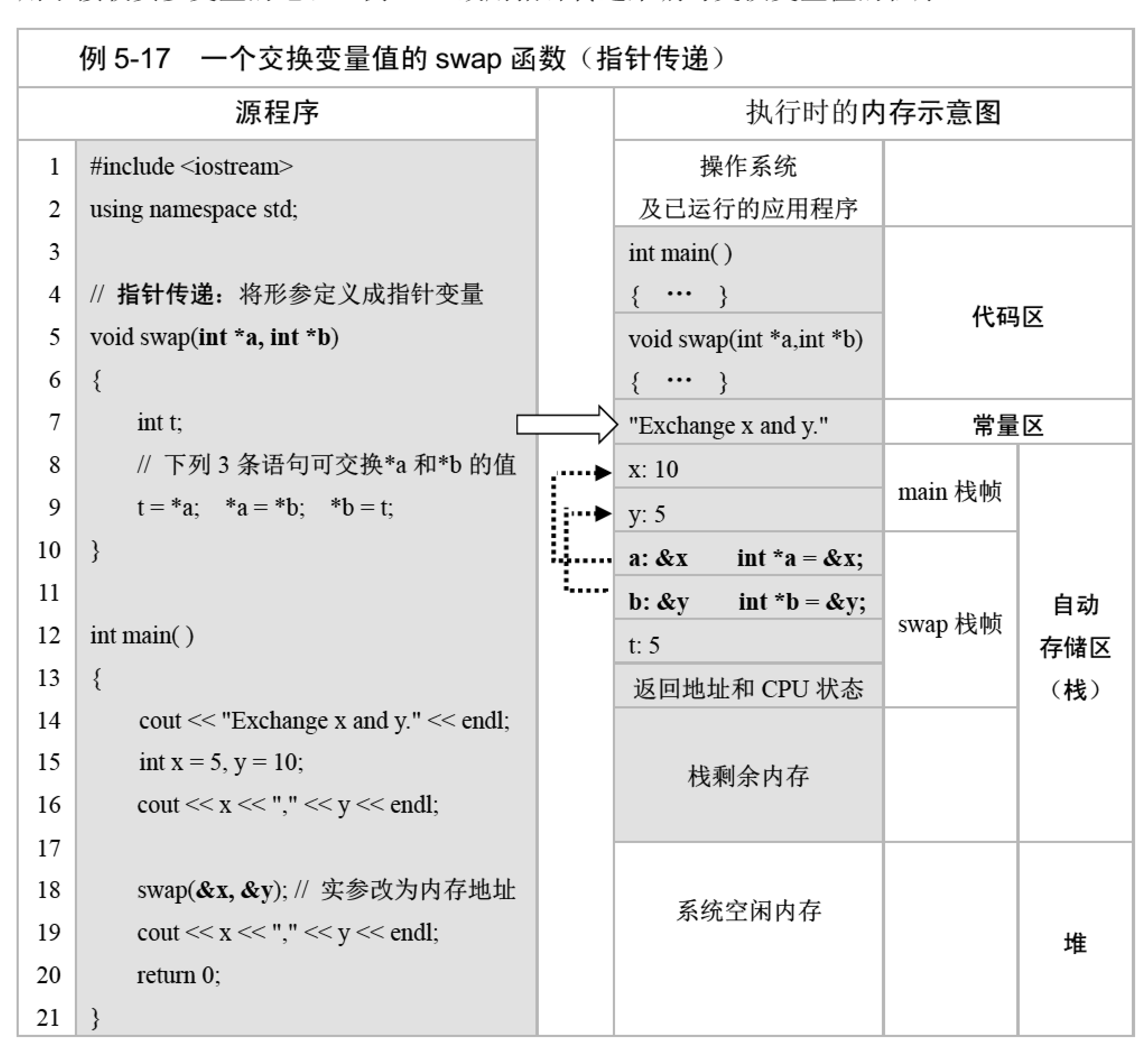
（2）被调函数修改形参实际上修改的是对应的实参。换句话说，主调函数可以通过引用将数据传给被调函数，被调函数也可以通过该引用修改实参的值，这等价于被调函数将修改后的数据传回主调函数。引用传递是一种双向数据传递机制。

（3）引用传递时，实参必须是变量。

（4）引用传递的好处，一是形参直接引用实参，不必另外分配内存，这样可以降低内存占用；二是可以通过引用实现函数间的双向数据传递。反过来，如果编写被调函数的程序员通过实参引用错误地修改了主调函数中实参变量的数据，这就会造成函数间互相干扰。

## 5.5.3 指针传递

**指针传递**就是传递实参变量的内存地址，被调函数通过该地址间接访问主调函数中的变量，从而达到传递数据的目的。采用指针传递时，被调函数的形参需定义成指针变量，用于接收实参变量的地址。例 5-17 改用指针传递来编写交换变量值的程序。


**例 5-17 一个交换变量值的 swap 函数（指针传递）**

源程序

```
#include <iostream>
using namespace std;

// 指针传递：将形参定义成指针变量
void swap(int *a, int *b)
{
      int t;
      // 下列 3 条语句可交换*a 和*b 的值
      t = *a;    *a = *b;    *b = t;
}

int main( )
{
      cout << "Exchange x and y." << endl;
      int x = 5, y = 10;
      cout << x << "," << y << endl;

      swap(&x, &y); // 实参改为内存地址
      cout << x << "," << y << endl;
      return 0;
}
```

例 5-17 中函数 swap 定义了两个 int 型指针形参 a、b（第 5 行），相应地主函数调用时所传递的两个实参分别是 x、y 的内存地址（第 18 行）。主函数调用 swap 函数，指针传递的参数传递过程相当于执行了下列语句：

```
int *a = &x;              // a 指向变量 x，后面访问*a 就是访问 x
int *b = &y;              // b 指向变量 y，后面访问*b 就是访问 y
```

函数 swap 交换*a、*b 的值（第 9 行），实际上就是交换了主函数中变量 x、y 的值。

例 5-17 的右侧给出了函数 swap 刚刚执行结束，准备返回主函数之前那个时刻的内存示意图。主函数在函数调用语句之后再次显示 x 和 y 的值，将显示 10 和 5。采用指针传递方式，函数 swap 成功交换了变量 x 和 y 的值。

指针传递具有如下特点：

（1）指针传递将主调函数中实参变量的内存地址传给被调函数的指针形参，被调函数通过该内存地址间接访问主调函数中的变量。

（2）被调函数可通过内存地址修改对应的实参。换句话说，主调函数可以通过内存地址将数据传给被调函数，被调函数也可以通过该地址修改实参的值，这等价于被调函数将修改后的数据传回主调函数。指针传递也是一种双向数据传递机制。

（3）指针传递时，实参必须是指针变量或指针表达式。

（4）指针传递的好处，一是可以通过内存地址实现双向数据传递，二是在传递批量数据（例如数组）时只需要传递一个内存首地址，这样可以提高数据传递效率，降低内存占用。与引用传递一样，如果编写被调函数的程序员通过内存地址错误修改了主调函数中的数据，这也会造成函数间互相干扰。

实际应用中，程序员应根据函数功能要求合理选择函数的参数及其传递方式，这就是函数参数的设计。

## 5.5.4　函数参数的设计

本节以一个具体的程序实例来讲解如何根据函数功能要求合理选择函数的参数及其传递方式。程序问题描述如下：

从键盘输入两个正整数 x 和 y，编写函数求出这两个数的最大公约数（greatest common divisor）和最小公倍数（least common multiple）。

让我们先来了解一下求最大公约数和最小公倍数的算法。

### 1. 算法设计

**定理：**将 a 和 b 的最大公约数记为 gcd(a, b)。假设 a≥b，并且 a÷b 的余数为 r，则 b 和 r 的最大公约数就是 a 和 b 的最大公约数，即 gcd(a, b) = gcd(b, r)。根据这个定理所设计出的求最大公约数算法称为**辗转相除法**。具体算法步骤如下：

（1）求 a 和 b 的最大公约数 gcd(a, b)，首先求 a÷b 的余数 r，然后判断余数 r 是否为 0。

（2）若余数 r = 0，则较小的那个数 b 就是所要求的最大公约数，即 gcd(a, b) = b，算法结束。

（3）若余数 r 不等于 0，则继续求两个数中较小的那个数 b 和余数 r 的最大公约数，即

继续求 gcd(b, r)。

（4）求 b 和 r 最大公约数与求 a 和 b 最大公约数的算法过程完全一样，即先求 b÷r 的余数，然后再判断余数是否为 0。重复上述过程，直到余数为 0，此时第二个变量 r 的数值即为所要求的最大公约数。

例如，假设 a = 25，b = 15，求这两个数最大公约数的过程如下：

（1）25÷15 的余数为 10。余数不等于 0，继续求 15 和 10 的最大公约数。

（2）15÷10 的余数为 5。余数不等于 0，继续求 10 和 5 的最大公约数。

（3）10÷5 的余数为 0，此时第二个数 5 即为 10 和 5 的最大公约数。根据上述最大公约数定理，5 也为上一步 15 和 10 的最大公约数。同理，5 也为再上一步 25 和 15 的最大公约数。

可以使用循环结构来实现上述辗转相除法。在使用辗转相除法求出最大公约数之后，求最小公倍数的算法比较简单。将 a 和 b 的最小公倍数记为 lcm(a, b)，其算法公式如下：

$$\mathrm{lcm}(a, b) = a\times b\div \mathrm{gcd}(a, b)$$

即：最小公倍数 = a×b÷最大公约数。例如，25 和 15 的最大公约数是 5，它们的最小公倍数等于 25×15÷5 = 75。

例 5-18 给出了求最大公约数和最小公倍数的完整 C++代码。

**例 5-18　求最大公约数和最小公倍数的 C++程序**

```
1   #include <iostream>
2   using namespace std;
3
4   int gcd(int a, int b)                    // 函数功能：求最大公约数
5   {
6       if (a <= 0 || b <= 0)   return 0;    // 如果 a、b 不是正整数，则直接退出
7
8       int r, t;                            // 定义一个保存余数的变量 r 和一个临时变量 t
9       if (a < b)                           // 辗转相除法要求 a≥b
10      {
11          t = a;   a = b;   b = t;         // 如果 a<b，则交换 a、b 的值
12      }
13      while ( true )                       // 使用循环结构实现辗转相除法
14      {
15          r = a % b;                       // 求 a 除以 b 的余数，保存到变量 r 中
16          if (r == 0)                      // 如果余数为 0，则 b 为最大公约数
17              return b;                    // 退出函数，返回 b 中保存的最大公约数
18          // 如果 r 不等于 0，则继续求 b 除以 r 的余数，即辗转相除
19          a = b;   b = r;                  // 将 b 赋值给 a，r 赋值给 b，转 13 行继续下一次循环
20      }
21  }
22
23  int lcm(int a, int b, int g)             // 函数功能：求最小公倍数
24  {
25          return a*b / g;                  // g 为 a 和 b 的最大公约数
26  }
27
```

```
28 int main( )
29 {
30         int x, y;
31         cin >> x >> y;                      // 从键盘输入 x 和 y 的值（应当为正整数）
32 
33         int xy_gcd, xy_lcm;                 // 定义两个变量分别保存最大公约数和最小公倍数
34         xy_gcd = gcd(x, y);                 // 调用函数 gcd 求 x 和 y 的最大公约数
35         xy_lcm = lcm(x, y, xy_gcd);         // 调用函数 lcm 求 x 和 y 的最小公倍数
36         // 显示结果
37         cout << x << "和" << y << "的最大公约数为" << xy_gcd << endl;
38         cout << x << "和" << y << "的最小公倍数为" << xy_lcm << endl;
39         return 0;
40 }
```

例 5-18 程序中，有两处语法细节需要做进一步说明。

（1）第 17 行。此处的 return 语句被包含在第 13 行 while 语句的循环体中。因为 while 语句将循环条件设为 true，即死循环，那么该如何结束循环呢？本例是通过 return 语句结束循环的。计算机执行 return 语句时将退出函数的执行，其所在的循环也会被自动停止。

（2）第 19 行。此处使用两条赋值语句来修改变量 a、b 的值。只有修改 a、b 的值，下一步再循环执行第 15 行的取余运算 a%b 时所求出的才是 b 除以 r 的余数。每次循环后变量 a、b 都被赋以新的值，这就是循环中变量的数值迭代。**数值迭代**是循环结构中一种基本的算法表现形式。

### 2. 参数设计

例 5-18 为函数 gcd 和 lcm 分别定义了两个形参 a、b，用于接收主函数所传递过来的实参数据 x 和 y，这是一种值传递方式。能不能将 gcd 和 lcm 这两个函数合并成一个函数，同时实现求最大公约数和最小公倍数的功能呢？答案是肯定的，但问题的难点在于如何将最大公约数和最小公倍数这两个计算结果同时返回给主函数。C++语言中，return 语句只能返回一个计算结果，因此每个函数也只能有一个返回值。前面我们介绍过，参数的引用传递或指针传递可以在主调函数和被调函数之间双向传递数据。利用这种双向数据传递机制，我们可以让一个函数返回多个计算结果。改写例 5-18 中函数 gcd 和 lcm，将它们合并成一个函数 gcd_lcm，其示意代码如下。

```
void gcd_lcm(int a, int b, int &rGCD, int *pLCM)      // 求最大公约数和最小公倍数
{
     if (a <= 0 || b <= 0)    return;                 // 如果 a、b 不是正整数，则直接退出
     int r, t;                                        // 定义一个保存余数的变量 r 和一个临时变量 t
     if (a < b)                                       // 辗转相除法要求 a≥b
     {
          t = a;    a = b;    b = t;                  // 如果 a<b，则交换 a、b 的值
     }
     // 先将 a 和 b 的积暂存在 t 中，因为后面求最小公倍数时需要用到
     t = a * b;
     while ( true )                                   // 使用循环结构实现辗转相除法
     {
```

```
        r = a % b;                      // 求 a 除以 b 的余数，保存到变量 r 中
        if (r == 0)                     // 如果余数为 0，则 b 为最大公约数
        {
            rGCD = b;                   // 将 b 中的最大公约数赋值给 rGCD
            break;                      // 跳出死循环
        }
        // 如果 r 不等于 0，则继续求 b 除以 r 的余数，即辗转相除
        a = b;   b = r;                 // 将 b 的值赋值给 a，r 的值赋值给 b
    }
    // 循环结束后，rGCD 中保存的是最大公约数
    *pLCM = t / rGCD;                   // 求最小公倍数（之前已经将 a 和 b 的积暂存在 t 中了）
    return;
}
```

修改后，主函数只需调用一次 gcd_lcm 函数，就可以同时求出最大公约数和最小公倍数。其调用形式如下：

```
gcd_lcm(x, y, xy_gcd, &xy_lcm);
```

函数 gcd_lcm 的代码说明如下：

（1）形参 a、b 继续采用值传递方式，单向接收主函数传递过来的实参数据 x 和 y。

（2）第 3 个形参 rGCD 是一个引用变量，调用时将引用主函数中对应的实参变量 xy_gcd。函数体中任何对形参 rGCD 的修改，修改的实际上是其引用的实参变量 xy_gcd。例如，将所求出的最大公约数赋值给 rGCD，实际上是赋值给了主函数中的变量 xy_gcd。调用结束后，主函数直接读取变量 xy_gcd 的数据就能得到最大公约数。采用引用传递，函数 gcd_lcm 完成了将第一个计算结果最大公约数传回主函数的任务。函数 gcd_lcm 可以继续采用引用传递将第二个计算结果最小公倍数传回主函数。为了演示语法，函数 gcd_lcm 刻意改用指针传递来完成这个任务。

（3）第 4 个形参 pLCM 是一个指针变量，调用时将指向主函数中对应的实参变量 xy_lcm。函数体中任何对“*pLCM”的修改，修改的实际上是其指向的实参变量 xy_lcm。例如，将所求出的最小公倍数赋值给“*pLCM”，实际上是赋值给了主函数中的变量 xy_lcm。调用结束后，主函数直接读取变量 xy_lcm 的数据就能得到最小公倍数。采用指针传递，函数 gcd_lcm 又完成了将第二个计算结果最小公倍数传回主函数的任务。

（4）函数 gcd_lcm 通过引用形参 rGCD 和指针形参 pLCM 分别将最大公约数和最小公倍数传回主函数，这已经完成了向主函数返回计算结果的任务。函数 gcd_lcm 不再需要通过函数返回值来返回数据，这时其函数类型应定义成 void，return 语句也不再需要带表达式。调用函数 gcd_lcm 时，主函数也不需要接收返回值，直接在函数调用后面加分号“;”就构成了一条完整的函数调用语句。

（5）在求最大公约数的循环中，函数 gcd_lcm 改用 break 语句来跳出循环。函数 gcd_lcm 没有继续使用 return 语句的原因是：在求出最大公约数之后不能立即退出函数，因为下面还要继续计算最小公倍数。

## 本节习题

1. 下列关于值传递的叙述，错误的是（　　）。
   A. 值传递只是将主调函数中实参的值传递给被调函数的形参
   B. 值传递时，实参可以是常量、变量或表达式
   C. 值传递是一种单向数据传递机制
   D. 被调函数对形参的修改将会影响到主调函数中的实参
2. 下列关于引用传递的叙述，错误的是（　　）。
   A. 引用传递将被调函数的形参定义成主调函数中实参变量的引用
   B. 引用传递时，实参可以是常量、变量或表达式
   C. 引用传递是一种双向数据传递机制
   D. 引用传递时，被调函数修改形参实际上修改的是所引用的实参
3. 下列关于指针传递的叙述，错误的是（　　）。
   A. 指针传递将主调函数中实参变量的内存地址传给被调函数的指针形参
   B. 指针传递时，实参必须是指针变量或指针表达式
   C. 指针传递是一种单向数据传递机制
   D. 指针传递时，被调函数可通过指针形参所接收到的实参地址修改实参
4. 函数间参数传递不能采用下列哪种方式？（　　）
   A. 值传递　　　　B. 引用传递
   C. 指针传递　　　D. 被调函数直接访问主调函数中的局部变量
5. 采用下列哪种方式不能实现函数间数据的双向传递？（　　）
   A. 值传递　　B. 引用传递　　C. 指针传递　　D. 共用全局变量
6. 已定义主调函数 fun 和被调函数 subfun：

```
void subfun( int a, int *b )   {   …   }
void fun( )
{
      int x = 5, y = 10;
      subfun( x, &y );
      // 其余代码省略
      }
```

   上述两个函数之间使用了什么参数传递方式？（　　）
   A. 值传递　　　　　　B. 引用传递
   C. 值传递和引用传递　D. 值传递和指针传递

# 5.6　在函数间传递数组

函数在函数体中定义的数组是局部变量，其他函数不能直接访问其中的数据，需要时可通过参数传递来实现对数组的共享。传递数组时，被调函数需定义数组形式的形参，同

时还需传递表明数组大小的参数，例如数组的元素个数。

## 5.6.1 在函数间传递一维数组

传递数组时，被调函数需定义数组形式的形参。调用函数时所对应的实参应是某个已定义数组的数组名。例 5-19 给出一个在函数间传递一维数组的 C++演示程序。

**例 5-19 在函数间传递一维数组的 C++演示程序（数组形式的形参）**

```
#include <iostream>
using namespace std;

int Max( int array[ ], int length )     // 求数组 array 中元素的最大值，length 表示该数组的长度
{
    int value = array[0], n;
    for (n = 1; n < length; n++)
    {
        if (array[n] > value)    value = array[n];
    }
    return value;
}

int main( )
{
    int a[5] = { 2, 8, 4, 6, 10 };          // 定义一维数组 a，定义时初始化
    cout << Max( a, 5 ) << endl;            // 调用函数 Max 求数组 a 中的最大值，结果为 10
    return 0;
}
```

例 5-19 程序的说明如下：

- 被调函数 Max。被调函数定义一维数组形式的形参，其语法形式为：

  **一维数组形参名[ ]**

  例如 array[ ]。被调函数还应当定义接收一维数组大小的形参，例如 length。
- 主调函数 main。主调函数在调用被调函数时应给出数据类型与形参数组相同的实参数组，以及该数组实参的大小，例如 Max( a, 5 )。其中实参数组 a 应是已经定义好的数组。

## 5.6.2 在函数间传递一维数组的首地址

计算机内部对数组的管理和访问是通过指针（即内存地址）来实现的。在函数间传递数组，所传递的实际上是数组的首地址。换句话说，函数间数组传递采用的是指针传递，而不是值传递。例 5-20 给出另一个传递一维数组的 C++演示程序。该例中被调函数的数组形参被显式地定义成一个指针变量，而函数调用语句中的实参保持不变。

**例 5-20 在函数间传递一维数组的 C++演示程序（指针形式的形参）**

```
1   #include <iostream>
2   using namespace std;
3   
4   int Max( int *pArray, int length )          // 求指针 pArray 所指向数组中元素的最大值
5   {
6       int value = *pArray, n;                 // *pArray 也可以写成下标形式：pArray[0]
7       for (n = 1; n < length; n++)
8       {
9           if (*(pArray+n) > value)    value = *(pArray+n);
10          // 上述语句也可以写成下标形式：if (pArray[n] > value)    value = pArray[n];
11      }
12      return value;
13  }
14  
15  int main( )
16  {
17      int a[5] = { 2, 8, 4, 6, 10 };          // 定义一维数组 a，定义时初始化
18      cout << Max( a, 5 ) << endl;            // 调用函数 Max 求数组 a 中的最大值，结果为 10
19      return 0;
20  }
```

在函数间传递数组这样的批量数据时，只传递数组首地址可以大幅提高数据传递效率。反过来，如果编写被调函数的程序员通过内存地址错误修改了主调函数中的数组数据，这也会造成函数间互相干扰。

假设例 5-20 的程序是由甲、乙两位程序员共同编写的。甲负责编写主函数 main，乙负责编写子函数 Max。如果编写子函数 Max 的程序员乙因为疏忽，打字时将第 9 行 if 语句条件中的“>”打成了“=”，错误写成了如下的形式：

```
if (*(pArray+n) = value)    value = *(pArray+n);          // 应该是：if (*(pArray+n) > value)
```

语法上，这条语句是正确的，程序编译可以正常通过。但执行这个程序，会发现主函数调用子函数 Max 求数组 a 中最大值所得到的结果为 2，这是错误的（正确结果应为 10）。更为严重的是，计算机内部在计算 if 语句中的条件“*(pArray+n) = value”时，会将 pArray 所指向数组 a 中所有元素的值都改为 2（即第 0 个元素的值，读者可以分析一下原因）。

编写主函数的程序员甲调用子函数 Max，其目的是要求数组 a 中的最大值。没想到不仅没有得到正确的结果，连已经初始化好的数组也被破坏了。当然，程序员乙不是故意的，那么他该如何避免这种情况的发生呢？

为避免上述情况的发生，程序员乙在编写子函数 Max 时可以为指针形参 pArray 添加一个 const 关键字，将其定义成指向常变量的指针。具体语法形式如下：

```
int Max( const int *pArray, int length )          // 形参 pArray 被定义成指向常变量的指针
{  …  }                                           // 函数体代码不变，省略
```

修改后，如果程序员乙因为疏忽将第 9 行 if 语句条件中的“>”输入成了“=”，那么在编译程序时编译器就会提示语法错误。因为通过指向常变量的指针只能读取所指向变量

中的值，而不能修改，否则属于语法错误，编译时就不能通过。语法上，定义指针变量时添加 const 关键字将其定义成指向常变量的指针，其含义是它所指向的变量是一个常变量（即只能读、不能改的变量）。C++编译器将负责检查语法，并向程序员提示所有违反上述语法规则的语句代码，这就是 C++语言所提供的 const **数据保护机制**。使用这种保护机制，程序员就不用再担心自己会无意中修改他人的数据，可以彻底避免程序员之间的相互干扰。

### 5.6.3 在函数间传递二维数组

程序员可以在函数间传递二维数组。例 5-21 给出一个在函数间传递二维数组的 C++演示程序。

**例 5-21 在函数间传递二维数组的 C++演示程序（数组形式的形参）**

```
#include <iostream>
using namespace std;

int Max( int array[ ][3], int row )     // 求数组 array 中元素的最大值，row 表示行数，3 是列数
{
    int value = array[0][0], m, n;
    for (m = 0; m < row; m++)
        for (n = 0; n < 3; n++)
        {
            if (array[m][n] > value)    value = array[m][n];
        }
    return value;
}

int main( )                             // 主函数
{
    int a[2][3] = { 2, 8, 4, 6, 10, 12 };    // 定义二维数组 a，定义时初始化
    cout << Max( a, 2 ) << endl;             // 调用函数 Max 求数组 a 中的最大值
    return 0;
}
```

## 学习本章的要点

- 读者要准确领会结构化程序设计的思想内涵，并熟练掌握 C++语言中函数相关的语法知识。
- 读者应深入计算机内部，了解程序执行时其代码和变量在内存中的存储原理，这样可以更容易理解变量作用域和生存期等抽象的概念。
- 读者要准确把握函数间传递数据的三种方式。
- 读者要分别从定义函数的程序员和调用函数的程序员这两个不同的角度去学习函数，这样才能更容易地理解函数相关的各种语法知识。

## 5.7　本章习题

1. **阅读程序**。阅读下列 C++程序。阅读后请说明各函数的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
using namespace std;
int fun(int x, int y)
{
    int z = x - y;
    if (z >= 0)    return z;
    else    return -z;
}
int main( )
{
    cout << fun( 6, 10 ) << endl;
    return 0;
}
```

2. **阅读程序**。阅读下列 C++程序。阅读后请说明各函数的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
using namespace std;
int fun(char str[ ])
{
    int n = 0, num = 0;
    while (str[n] != '\0')
    {
        if (str[n] >= 'A' && str[n] <= 'Z' || str[n] >= 'a' && str[n] <= 'z')    num++;
        n++;
    }
    return num;
}
int main( )
{
    cout << fun( "123 abc ABC" ) << endl;
    return 0;
}
```

3. **程序改错**。阅读下列 C++程序，并检查其中的语法错误。修改错误，并保证程序的功能不变。

```
#include <iostream>
using namespace std;
int main( )
{
    cout << fun( 10, 6.5 ) << endl;                    // 调用函数 fun 求 6.5÷10 的结果
    return 0;
```

```
}
void fun(double x, double y)                    // 定义求 x÷y 的函数 fun
{
    int z = x / y;
    return z;
}
```

4. **程序改错**。阅读下列 C++程序，并检查其中的语法错误。修改错误，并保证程序的功能不变。

```
#include <iostream>
using namespace std;
double Average( int array, int length )         // 定义求数组 array 中元素平均值的函数
{
    int n, sum;
    for (n = 1; n <= length; n++)               // 求所有元素的累加和
        sum += array[n];
    return ( sum/length );                      // 返回平均值
}
int main( )
{
    int *a, n;
    new a[10];                                  // 动态分配包含 10 个元素的 int 型数组
    for (n = 0; n < 10; n++)                    // 数组元素的值等于其下标的平方
        a[n] = n * n;
    cout << Average( a[10] ) << endl;           // 调用函数 Average 求其元素的平均值
    delete a;                                   // 释放动态分配的数组
    return 0;
}
```

5. **编写程序**。采用数据分散管理策略和集中管理策略，分别编写一个求长方形面积的 C++程序。要求：用子函数 Area 计算长方形面积，主函数 main 负责输入长宽数据、调用子函数 Area 求面积，并显示面积的计算结果。

6. **编写程序**。采用数据分散管理策略时，函数间的参数传递有三种方式，分别是值传递、引用传递和指针传递。请使用引用传递和指针传递，分别编写一个求长方形面积的 C++程序。要求：用子函数 Area 计算长方形面积，主函数 main 负责输入长宽数据、调用子函数 Area 求面积，并显示面积的计算结果。




# 第6章 结构化程序设计之二

一个 C++程序可以包含很多个函数，源代码可能会很长，可以将这些函数分散保存在不同的源程序文件中，以**多文件结构**的形式来组织和管理源代码。

程序员可以在 C++源程序中插入一些特殊指令，其作用是告诉编译器该如何编译本程序。正式编译源程序之前，编译器将预先处理这些特殊指令，因此它们被称为**编译预处理指令**。

本章还将介绍 C++语言中几种特殊形式的函数，以及由 C++编译系统为程序员提供的一些常用函数（称为**系统函数**）。

程序设计可能会面临比较复杂的数据，这时程序员需要基于基本数据类型来自己定义新的数据类型，这就是**自定义数据类型**。数组就是一种自定义数据类型，本章将再介绍几种常用的自定义数据类型。

本章最后以微软公司开发的 Win32 API 函数库为例介绍如何开发一个 Windows 图形用户界面程序，并对结构化程序设计方法进行简单的回顾和总结。

## 6.1 C++源程序的多文件结构

如果一个 C++程序包含多个函数，程序员可以将它们保存在不同的程序文件中，以多文件结构的形式来组织和管理程序源代码。

结构化程序设计方法将一个复杂程序设计任务分解成多个算法模块，编码时将每个模块定义成一个函数。基于团队分工协作开发程序时，可以将不同模块的设计和编码任务交给不同程序员去完成。程序员各自独立编程，将所编写的程序代码分别保存在自己的程序文件中，这样可以互不干扰。因此，基于团队分工协作所开发的 C++程序会自然形成多文件结构。

### 6.1.1 多文件结构的源代码组织

一个 C++程序开发**工程**（project）可以包含多个源程序文件，一个**源程序文件**（.cpp）可以包含多个函数。一个函数只能集中放在一个源程序文件中，不能将其定义代码拆开存放在不同文件中。一个程序开发工程可以包括很多函数，但只有一个主函数，其函数名必须为 main。

使用编译器将 C++语言翻译成机器语言。编译时，同一源程序文件中的所有函数被统一编译，因此一个源程序文件称为是一个**编译单元**。一个 C++程序开发工程可以包含多个源程序文件，每个源程序文件编译后都生成一个目标程序文件（.obj）。**目标程序**是机器语言的程序，通常不同厂家 CPU 的机器语言是不一样的。相同的 C++源程序可被不同的编译器编译，生成不同机器语言的目标程序，运行于不同的 CPU 之上。

一个多文件结构的 C++程序会编译出多个目标程序。使用**连接器**将多个目标程序连接在一起，生成一个可执行程序文件（Windows 操作系统上其扩展名为.exe）。**可执行程序**是最终的程序，可以被 CPU 识别和执行。软件产品销售的是可执行程序，而源程序则是软件厂家的机密。可执行程序文件可以复制分发、安装执行，但很难被阅读、修改。

在多文件结构中，一个文件中定义的函数可以被其他文件中的函数调用。可被其他文件调用的函数称为**外部函数**。一个文件中定义的全局变量也可以被其他文件中的函数访问。可被其他文件访问的全局变量称为**外部全局变量**。换句话说，一个文件中的函数可以调用其他文件中的外部函数，也可以访问其他文件中的外部全局变量。调用外部函数需要“先声明，再调用”，访问外部全局变量也需要“先声明，再访问”。声明的作用是将外部函数或外部全局变量的作用域延伸到本文件中来。

下面以第 5 章的测算养鱼池工程总造价程序为例，具体讲解多文件结构的源代码组织。该程序共有 3 个模块，其中一个主模块，两个子模块。假设由程序员甲和乙组成的两人小组来共同开发，并且继续采用例 5-10 所示的数据集中管理策略。甲是组长，负责定义全局变量并编写主函数，所编写的源程序存放在文件 1.cpp 中。乙是组员，负责编写两个子函数，所编写的源程序存放在文件 2.cpp 中。最终的测算养鱼池工程总造价程序包含两个源程序文件，即多文件结构，如例 6-1 所示。

**例 6-1　测算养鱼池工程总造价的 C++程序（多文件结构）**

**源程序文件 1.cpp**：包括全局变量和主函数的定义，由程序员甲编写

```
#include <iostream>
using namespace std;

void RectCost( );                  // 声明 2.cpp 中的外部函数：计算长方形养鱼池造价的函数
double CircleCost(double r);       // 声明 2.cpp 中的外部函数：计算圆形水池造价的函数

double length, width;              // 全局变量：分别保存长方形养鱼池的长和宽
double r1, r2;                     // 全局变量：分别保存圆形清水池和污水池的半径
double totalCost = 0;              // 全局变量：保存最终的计算结果，即总造价

int main( )                        // 主函数
{
    // 下列语句将从键盘输入的原始数据直接存放到对应的全局变量中
    cout << "请输入长方形的长和宽：" ;
    cin >> length >> width;
    cout << "请输入清水池和污水池的半径：" ;
    cin >> r1 >> r2;
    RectCost( );                   // 调用外部函数 RectCost 计算长方形养鱼池造价
    totalCost += CircleCost( r1 );       // 调用外部函数 CircleCost 计算圆形清水池造价
```

```
20      totalCost += CircleCost( r2 );          // 再次调用函数 CircleCost 计算圆形污水池造价
21      cout << "工程总造价为 " << totalCost << endl;
22      return 0;
23  }
```

**源程序文件 2.cpp**：包括两个子函数的定义，由程序员乙编写

```
1   extern double length, width; // 声明 1.cpp 中的外部全局变量：分别保存长方形养鱼池的长和宽
2   extern double totalCost;      // 声明 1.cpp 中的外部全局变量：保存最终的计算结果，即总造价
3
4   void RectCost( )              // 函数定义：计算长方形养鱼池造价
5   {
6       double cost ;
7       cost = length * width * 10 ;  // 直接读取全局变量 length 和 width 中的长宽数据
8       totalCost += cost ;           // 将计算结果直接累加到全局变量 totalCost 中
9   }
10
11  double CircleCost(double r)       // 函数定义：计算圆形水池造价
12  {
13      double cost ;
14      cost = 3.14 * r * r * 10 ;
15      return cost ;
16  }
```

例 6-1 程序的说明如下：

- **与例 5-10 的比较**。例 6-1 和例 5-10 所完成的程序功能完全相同，只是文件结构不同。例 5-10 是单文件结构，所有的函数定义和全局变量定义都在同一源程序文件中。例 6-1 是多文件结构，1.cpp 包含全局变量定义和主函数 main 的定义，2.cpp 包含两个子函数的定义。改用多文件结构后，函数或全局变量的定义形式没有任何变化，与例 5-10 完全一样。
- **增加外部函数的原型声明**。1.cpp 文件中的主函数需要调用 2.cpp 中的两个子函数 RectCost 和 CircleCost，因此 1.cpp 增加了对这两个外部函数的原型声明语句（1.cpp 中的第 4、第 5 行）。
- **增加外部全局变量的声明**。2.cpp 文件中的函数 RectCost 在计算长方形养鱼池造价时需访问全局变量 length 和 width，所计算出的造价也要累加到全局变量 totalCost 上。这 3 个变量都是定义在 1.cpp 中的外部全局变量，因此 2.cpp 增加了对它们的外部声明语句（2.cpp 中的第 1、2 行）。需要注意的是，2.cpp 并没有访问 1.cpp 中定义的全局变量 r1 和 r2，因此不需要声明这两个变量。

声明外部函数原型和外部全局变量的语法细则如下：

（1）**声明外部函数原型**。声明时可以使用关键字 **extern**，也可以不用，两者的语法作用完全相同。例如 1.cpp 中的第 4、第 5 行可以修改为：

```
extern void RectCost( );              // 声明外部函数：计算长方形养鱼池造价的函数
extern double CircleCost(double r);   // 声明外部函数：计算圆形水池造价的函数
```

使用“extern”关键字的作用是更明确地指出所声明的函数是一个外部函数。

（2）**声明外部全局变量**。声明时必须加上关键字 **extern**，另外不能初始化，否则就变成了另一条全局变量定义语句，连接时会出现两个全局变量重名的错误。例如 2.cpp 中第 2 行的声明语句不能简单复制其原来的定义语句（1.cpp 中的第 9 行）：

```
extern double totalCost = 0;                    // 错误：声明外部全局变量不能初始化
```

（3）**可以只声明，不使用**。外部函数可以只声明，不调用。外部全局变量也可以只声明，不访问。“只声明，不使用”不会引起语法错误。

## 6.1.2 静态函数与静态变量

多文件结构中，源程序文件中所定义的函数默认都是外部函数，可以被其他文件中的函数调用；所定义的全局变量默认都是外部全局变量，可以被其他文件中的函数访问。外部函数和外部全局变量被多文件结构中的所有文件共享，其他文件只要经过声明就都可以使用这些函数和全局变量。

多文件结构中，所有的外部函数不能重名，所有的外部全局变量不能重名，外部函数和外部全局变量之间也不能重名。

某些情况下，可能需要定义一些只在本文件内部使用的函数或全局变量。例如某个程序员在编写程序时，希望定义一些只供自己使用的子函数或全局变量。从设计和功能的角度，这些只供文件内部使用的函数或全局变量不应该，也不需要被其他文件中的函数访问，此时程序员可以使用关键字 static 将它们定义成**静态函数**或**静态全局变量**。另外，C++语言还可以定义**静态局部变量**。

### 1. 静态函数

定义函数时在函数头前面加关键字 static，该函数就被定义为静态函数。静态函数只能被本文件内部的其他函数调用，其他文件不能调用，即使经过声明也不行。例如，如果将例 6-1 中 2.cpp 第 4 行的 RectCost 函数定义改为：

```
static void RectCost( )            // 定义静态函数：计算长方形养鱼池造价
```

则 RectCost 函数就是一个静态函数。即使在 1.cpp 中对该函数进行声明，主函数也不能调用，否则程序连接时会提示错误。

关键字 static 将所定义的函数作用域限定在本文件范围内，禁止延伸到其他文件。合理定义静态函数，可以防止其他文件对该函数的误调用。不同文件中的静态函数可以重名。

### 2. 静态全局变量

定义全局变量时，在语句前面加关键字 static，该变量就被定义为静态全局变量。静态全局变量只能被本文件内的函数访问，其他文件不能访问，即使经过声明也不行。例如，例 6-1 中 1.cpp 所定义的两个全局变量 r1 和 r2（第 8 行）不需要被 2.cpp 访问，可以改为静态定义：

```
static double r1, r2;              // 静态全局变量：分别保存圆形清水池和污水池的半径
```

修改后程序的编译连接正确，没有错误。因为 2.cpp 中函数 CircleCost 计算圆面积所需的半径是通过形实结合获得的，不需要直接访问全局变量 r1 和 r2。

关键字 static 将所定义的全局变量作用域限定在本文件范围内，禁止延伸到其他文件。合理定义静态全局变量，可以防止其他文件对该变量的误操作。不同文件中的静态全局变量可以重名。

### 3. 静态局部变量

全局变量具有文件作用域，非静态、静态是指它们能否被其他文件中的函数访问。定义在函数里的局部变量具有块作用域，只能在函数内部访问。局部变量在任何情况下都不能被其他函数访问，更别说其他文件中的函数了。C++语言中局部变量也有静态、非静态之分，但含义与全局变量的静态与非静态是不一样的。

全局变量在源程序中具有文件作用域，程序执行时立即分配内存，一直到程序执行结束才被释放。全局变量存放在静态存储区。局部变量具有块作用域，程序执行后，当执行到其定义语句时才分配内存，到其所在代码块执行结束立即被释放。局部变量存放在自动存储区（即栈中）。

**静态局部变量**具有块作用域（与局部变量相同），程序执行时立即分配内存，存放在静态存储区，直到程序执行结束才被释放（与全局变量相同）。静态局部变量是居于全局变量和局部变量之间的一种折中变量。在源程序中，静态局部变量的作用域与局部变量相同。程序执行时，静态局部变量的内存生存期和存放位置与全局变量一样。在函数体或复合语句中定义局部变量时，在语句前面加关键字 static，该变量就被定义为静态局部变量。

例 6-2 给出一个对比静态和非静态局部变量的 C++演示程序。函数 fun 定义了两个局部变量，x 为普通局部变量（非静态），而 y 为静态。除了静态和非静态的区别之外，对这两个变量的其他操作完全相同。函数 fun 被主函数调用两次，每次调用时都会显示 x 和 y 的值。比较这两次显示结果的变化，就可以看出静态局部变量和非静态局部变量之间的区别。

**例 6-2　对比静态和非静态局部变量的 C++演示程序**

```
#include <iostream>
using namespace std;

void fun( )
{
    int x = 0;                    // 普通局部变量（非静态）x，初始化为 0
    static int y = 0;             // 静态局部变量 y，也初始化为 0
    x++;   y ++;                  // 对 x、y 做相同的加 1 操作
    cout << x << " and " << y << endl;
}

int main( )
{
    fun( );      // 调用函数 fun，其中所显示 x 和 y 的值都是 1
    fun( );      // 再次调用函数 fun，其中所显示 x 的值仍为 1，而 y 的值变为 2
    return 0;
}
```

例 6-2 程序的说明如下：

■ **普通局部变量** x（**非静态**）：变量 x 定义在函数 fun 内部，是局部变量，具有块作用域。执行该程序，主函数调用函数 fun，当执行到第 6 行的定义语句时为变量 x 从栈中分配内存并初始化为 0，加 1 操作后变为 1。此时显示 x 的值将会显示 1，如图 6-1(a)所示。函数 fun 执行结束返回主函数时，计算机会释放其栈帧，其中变量 x 的存储单元也被释放，所保存的数据 1 也被丢弃。主函数再次调用函数 fun，将会重复上述过程，为 x 分配内存并初始化为 0，加 1 后的显示结果仍为 1，如图 6-1(b)所示。

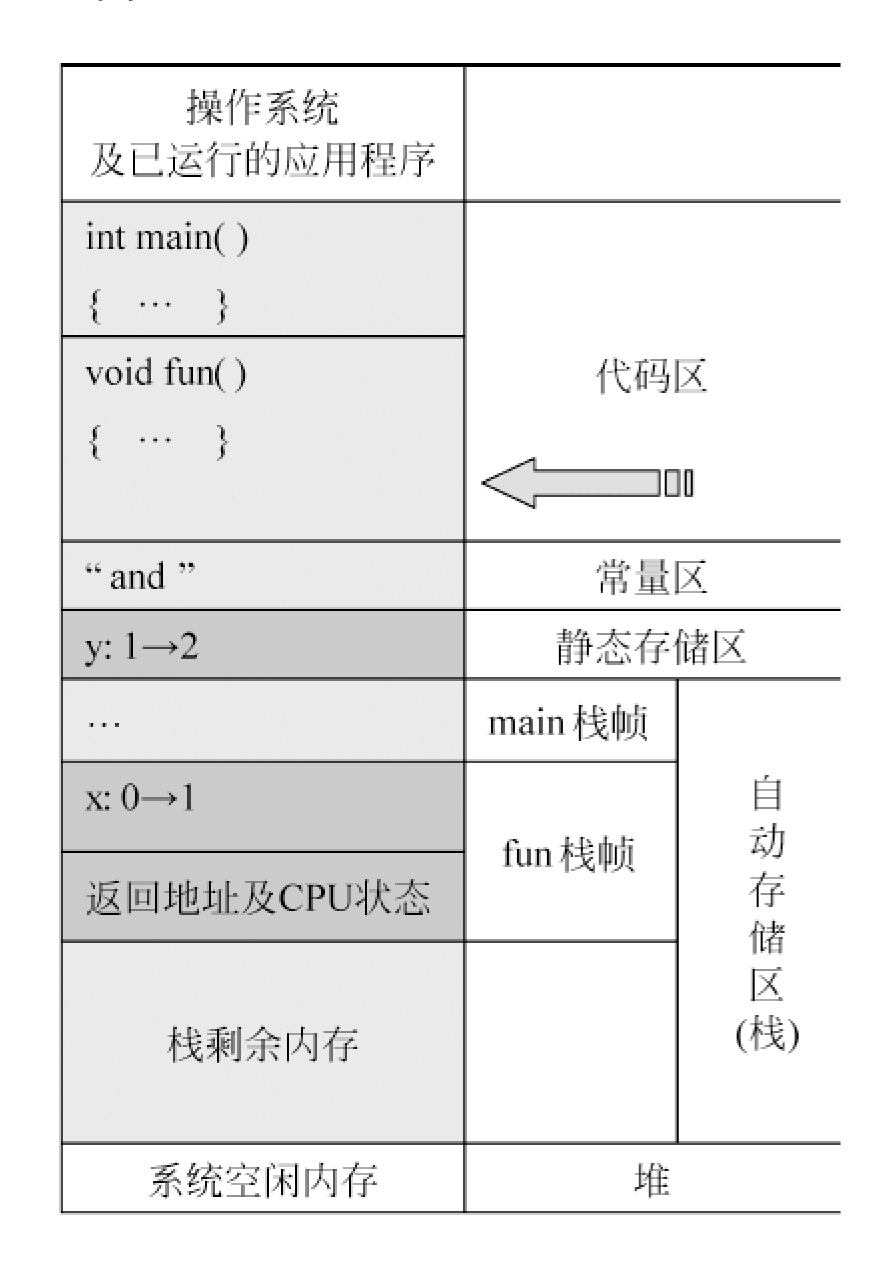
可以看出，如果函数被多次调用，其中的普通局部变量不能保留上次调用结束时的值。因为每次调用时普通局部变量都会重新分配内存单元，重新初始化。

■ **静态局部变量** y：变量 y 也定义在函数 fun 内部，是局部变量，具有块作用域，但定义时被加上了关键字 static，是静态局部变量。静态局部变量 y 在程序执行一开始即被分配内存并初始化为 0，这相当于是第 7 行的定义静态变量语句被提前执行了。静态局部变量 y 被存放在自动存储区，一直到程序执行结束才被释放。主函数调用函数 fun 时，第 7 行的定义静态变量语句会被跳过，不再执行。第一次调用函数 fun 时，y 的初始值为 0，然后被加 1，如图 6-1(a)所示。函数 fun 执行结束返回主函数时，计算机会释放其栈帧，变量 x 的存储单元将被释放。但变量 y 的存储单元在静态存储区，不会被释放，其中的数据 1 也会保留下来。主函数第二次调用函数 fun，再次将 y 加 1，y 的值变为 2，如图 6-1(b)所示。





(a) 第1次调用函数fun时的程序副本




(b) 第2次调用函数fun时的程序副本

图 6-1 例 6-2 程序执行时的内存示意图

可以看出，函数中的静态局部变量定义语句只会在程序一开始被执行一次，后面不会再重复执行。如果函数被多次调用，其中的静态局部变量可以保留上次调用结束时的值，因为它的内存单元是在程序执行一开始即被分配并初始化，直到程序执行结束才被释放，其中的数据也会一直保留下来。本例中，每次调用函数 fun 所显示出的静态变量 y 的值，等于函数 fun 被调用的次数。

仔细分析可以发现，静态局部变量实际上是一种只能被某个特定函数访问的全局变量。如果程序员希望全局变量只能被某个特定的函数访问，那么可以将全局变量定义成该函数内部的静态局部变量。

使用关键字 static 定义静态全局变量和静态局部变量，这两个场合中“静态”的含义是有区别的。静态全局变量中“静态”的作用是将变量的作用域限定在某个源程序文件内部，而静态局部变量中“静态”的作用是将变量的作用域限定在某个函数内部。可以看出，C++语言中的 static 是一个多义词，这一点请读者注意。

### 6.1.3　头文件

团队分工协作开发程序时，某个程序员甲编写了一个 C++源程序文件（假设为 1.cpp），假设其中定义了一组函数，也定义了若干全局变量，这些函数和全部变量是提供给项目组其他程序员使用的。

另一个程序员乙在编写程序时需要调用 1.cpp 中的函数，或访问其中的全局变量，就需要在自己的源程序文件（假设为 2.cpp）中对这些函数或全局变量进行声明。访问多少个外部函数或全局变量，就需要编写多少条声明语句。项目组的所有程序员，只要访问 1.cpp 中的函数或全局变量，就都需要在自己的程序文件中编写声明语句。编写这些声明语句是重复而又枯燥的工作，为此 C++语言引入了**头文件**（header file）的概念。

程序员甲在编写好 1.cpp 源程序文件后，另外再编写一个头文件，其中包含 1.cpp 中所有外部函数和外部全局变量的声明语句。习惯上将该头文件命名为 1.h，即与源程序文件同名，扩展名为.h 或.hpp。如果项目组的其他程序员需要访问 1.cpp 中的外部函数或全局变量，只要在自己的程序文件中增加如下一条指令就可以实现声明的功能。

```
#include "1.h"
```

这条指令的作用是将头文件 1.h 中的所有声明语句自动插入到该指令所在位置，这就省去了以往一条一条手工编写声明语句的烦琐。插入的头文件可能包含多个外部函数或全局变量的声明，其中一些会被使用，而另一些不会。C++语言允许“只声明，不使用”。

应用头文件虽然增加了程序员甲的工作量，但能大大减轻项目组其他程序员编写声明语句的工作量。在掌握头文件的语法知识后，可以对例 6-1 的程序进行改写，如例 6-3 所示。例 6-3 和例 6-1 所完成的程序功能完全相同，只是引入了头文件。

**例 6-3　测算养鱼池工程总造价的 C++程序（应用头文件）**

**源程序文件 1.cpp**：由程序员甲编写

```
#include <iostream>
using namespace std;

#include "2.h"    // 用头文件 2.h 取代下面的声明语句
void RectCost( );
double CircleCost(double r);    // 删除上面这 2 条声明语句

double length, width;
static double r1, r2;
double totalCost = 0;

int main( )                        // 主函数
{
    cout << "请输入长方形的长和宽：" ;
    cin >> length >> width;
    cout << "请输入清水池和污水池的半径：" ;
    cin >> r1 >> r2;
    RectCost( );
    totalCost += CircleCost( r1 );
    totalCost += CircleCost( r2 );
    cout << "工程总造价为 " << totalCost << endl;
    return 0;
}
```

**头文件 1.h**：由程序员甲编写

```
/*
本头文件声明了 1.cpp 中的
3 个外部全局变量
*/

extern double length, width;
extern double totalCost;
```

**源程序文件 2.cpp**：由程序员乙编写

```
#include "1.h"    // 用头文件 1.h 取代下面的声明语句
extern double length, width;
extern double totalCost;    // 删除上面这 2 条声明语句

void RectCost( )            // 计算长方形养鱼池造价的函数
{
    double cost ;
    cost = length * width * 10 ;
    totalCost += cost ;
}

double CircleCost(double r)    // 计算圆形水池造价的函数
{
    double cost ;
    cost = 3.14 * r * r * 10 ;
    return cost ;
}
```

**头文件 2.h**：由程序员乙编写

```
/*
本头文件声明了 2.cpp 中的
2 个外部函数
*/

void RectCost( );
double CircleCost(double r);
```

例 6-3 程序的说明如下：

**头文件 1.h**。程序员甲编写的 1.cpp 中有 3 个全局变量需要被 2.cpp 访问，即 length、width 和 totalCost。为此，程序员甲再编写一个头文件 1.h，其中包含对这 3 个全局变量的声明语句。

2.cpp 中的函数 RectCost 在访问 1.cpp 中的这 3 个全局变量之前需要先声明，然后才能访问。为此，程序员乙在文件 2.cpp 的开头编写了一条插入头文件指令：

```
#include "1.h"
```

这条指令的作用是将头文件 1.h 中的两条全局变量声明语句插入到当前位置，从而实现了对上述 3 个全局变量的声明。1.cpp 中的另外两个全局变量 r1 和 r2 不需要被 2.cpp 访问，可以将它们定义成静态全局变量，1.h 中也不需要对它们进行声明。

**头文件 2.h**。类似地，程序员乙编写的 2.cpp 中有两个函数需要被 1.cpp 中的主函数调用，即 RectCost 和 CircleCost。为此，程序员乙再编写一个头文件 2.h，其中包含对这两个函数的声明语句。这时程序员甲在编写 1.cpp 时就不再需要手工编写函数声明语句，而是在程序开头简单地使用了如下的插入头文件指令：

```
#include "2.h"
```

这样就能自动插入头文件 2.h 中对上述两个函数的声明语句。

例 6-3 中的两位程序员在编写完自己的程序代码之后，又各自编写一个头文件来声明自己的外部函数或全局变量。对程序员来说，编写头文件就是辛苦自己，方便他人。C++语言中的头文件是一种充满了正能量的语法，体现了以程序员为本的思想。

头文件的内容主要包括外部函数的声明、外部全局变量的声明，还包括一些公共的符号常量定义和数据类型定义（将在后续章节中讲解）等。头文件中还可以再插入其他头文件。插入头文件所使用的“#include”指令是一种特殊指令，被称为编译预处理指令。下一节将具体讲解编译预处理指令。

## 本节习题

1. 关于 C++源程序文件，下列叙述不正确的是（　　）。
   A. 源程序文件主要用于存放函数的定义代码
   B. 在一个源程序文件中可以定义多个函数
   C. 一个函数定义代码可以分散保存在多个源程序文件中
   D. 函数可以调用其他源程序文件中的函数，但需对被调函数进行声明
2. 关于 C++程序文件，下列叙述不正确的是（　　）。
   A. 一个 C++程序可能有多个程序文件
   B. 程序文件主要由源程序文件和头文件组成
   C. 一个 C++程序可能由多个函数组成，但只能有一个 main 函数

D. 在一个源程序文件中只能定义一个函数

3. 下列哪个函数不能被其他源程序文件中的函数调用？（　　）

A. int fun(int x)　{ … }　　B. extern int fun(int x)　{ … }

C. void fun(int x)　{ … }　　D. static void fun(int x)　{ … }

4. 关于 C++头文件，下列叙述不正确的是（　　）。

A. 头文件主要用于对外部全局变量、外部函数和符号常量等进行声明

B. 可使用#include 指令将头文件插入到源程序文件中

C. 一个 C++程序只能有一个头文件

D. 头文件可以减少程序员编写声明语句的工作量

5. 函数 fun 中定义了一个局部变量 x：

```
void fun( )
{
    static int  x;
    …
}
```

假设程序执行过程中，函数 fun 被调用了 3 次，则变量 x 经历了几次内存分配－释放的过程？（　　）

A. 0　　B. 1　　C. 2　　D. 3

## 6.2 编译预处理指令

程序员可以在 C++源程序中插入一些特殊指令，其作用是告诉编译器该如何编译本程序。正式编译源程序之前，编译器将预先处理这些特殊指令，它们被称为**编译预处理指令**，例如插入头文件所使用的“#include”就是一种编译预处理指令。编译器在处理完所有的编译预处理指令之后才进行正式编译。

编译预处理指令不属于 C++语言的主体，是其附属组成部分，其作用是方便程序员使用 C++语言。常用的编译预处理指令有三种，它们分别是文件包含指令、宏定义指令和条件编译指令。

在 C++源程序中，编译预处理指令可以写在代码的任意位置，每条指令单独写一行，必须以井号“#”开头，不加分号“;”结束符。

### 6.2.1 文件包含指令

编写 C++源程序时，程序员可以使用**文件包含指令**（#include）将某个指定文件的内容插入到程序代码的当前位置，通常是用于将某个头文件（.h）插入到源程序文件（.cpp）中。文件包含指令是一条编译预处理指令。预处理时，编译器会将所指定的文件内容插入到指令所在的代码位置。

**编译预处理指令语法：文件包含指令**

```
#include <文件名>
或
#include "文件名"
```

语法说明：

- **文件名**通常不写完整的全路径，而使用默认路径，即只写头文件的文件名。通常头文件被集中存放在两个目录下：一是 C++编译器安装目录下的 Include 子目录，该目录称为标准目录；二是源程序文件所在的目录，称为当前目录。
- #include <文件名>：缺省路径时，编译器将到标准目录下搜索指定的文件。
- #include "文件名"：缺省路径时，编译器将首先在当前目录下搜索，如果搜索不到指定的文件，再去标准目录下搜索。
- 文件包含指令以井号“#”开头，单独写一行，不加分号“;”结束符。

例如本章 6.1.3 节例 6-3 所示的 C++程序，源程序文件 1.cpp 中分别使用了两条文件包含指令：

**1）第 1 行：#include <iostream>**

编译器处理这条文件包含指令时，将文件 iostream 中的内容插入到 1.cpp 第 1 行的代码位置。由于采用默认路径，编译器将到其安装目录下的 Include 子目录中搜索该文件。文件 iostream 是 C++编译系统提供的头文件，凡使用标准输入流 cin 和标准输出流 cout 输入/输出数据的程序都需要包含该头文件。

**2）第 4 行：#include "2.h"**

编译器处理这条文件包含指令时，将文件 2.h 中的内容插入到 1.cpp 第 4 行的代码位置。由于采用默认路径，编译器将在当前目录下搜索该文件，当前目录就是源程序文件 1.cpp 所在的目录。2.h 是程序员自己编写的头文件。一个程序开发项目在管理程序文件时，通常将项目组自己编写的头文件与其源程序文件放在同一目录下进行集中管理。

## 6.2.2　宏定义指令

C++源程序中，允许用一个标识符来表示一段代码文本，这就称为一个**宏**（macro）。其中的标识符称为**宏名**，所表示的代码文本称为**宏文本**。宏需要“先定义，再使用”。程序员编写程序时，使用**宏定义指令**（#define）来定义一个新的宏，这样其后续代码中凡是需要宏文本的地方都可以用宏名来代替。宏可以使源程序更加简洁，易于阅读。

宏定义指令是一条编译预处理指令。预处理时，编译器将源程序中所有的宏名自动替换回原来的宏文本，这称为**宏替换**或**宏展开**。C++语言有三种形式的宏，分别为无参宏、有参宏和空宏。已定义的宏可以用宏删除指令（#undef）删除。

### 1. 无参宏

**编译预处理指令语法：无参宏定义指令**

```
#define 宏名 宏文本
```

语法说明：

- **宏名**需符合标识符的命名规则，习惯上用大写字母来命名。
- **宏文本**指定宏名所表示的字符串，可以由任意字符组成，不需要加引号。

无参宏主要用于定义符号常量。例 6-4 是一个计算圆面积和周长的程序，代码第 4 行使用一条无参宏定义指令为 π 定义一个符号常量，其中宏名为 PI，宏文本为 3.141 59。后续代码第 11、12 行计算原面积和周长时需要用到 3.141 59，这时可以用 PI 来代替。

**例 6-4　应用宏定义指令的 C++演示程序（无参宏）**

```
1   #include <iostream>
2   using namespace std;
3
4   #define   PI   3.14159                        // 为 π 定义一个符号常量
5
6   int main( )                                   // 主函数
7   {
8       cout << "请输入圆的半径：" ;
9       cin >> r;
10
11      cout << "面积：" << PI * r * r << endl ;   // 用符号常量 PI 表示 3.141 59
12      cout << "周长：" << PI * 2 * r << endl ;   // 用符号常量 PI 表示 3.141 59
13      return 0;
14  }
```

编译器在编译上述 C++程序时首先进行预处理，将源程序中所有的宏名 PI 自动替换回原来的宏文本 3.141 59，然后再进行正式编译。例如代码第 11、12 行预处理后的结果如下：

```
cout << "面积：" << 3.14159 * r * r << endl;
cout << "周长：" << 3.14159 * 2 * r << endl;
```

编译器对这两条语句进行正式编译，将它们翻译成机器语言。

### 2. 有参宏

**编译预处理指令语法：有参宏定义指令**

```
#define 宏名(参数列表) 宏文本
```

语法说明：

- **宏名**需符合标识符的命名规则，习惯上用大写字母来命名。
- **参数列表**指定若干可被替换的参数，参数之间用逗号“,”隔开。
- **宏文本**指定宏名所表示的字符串，可以由任意字符组成，不需要加引号。字符串通常是包含参数的表达式。

有参宏可以实现简单的函数功能。例如，如果定义如下计算圆面积的有参宏：

```
#define   AREA(x)   3.14159 * x * x                // 定义时的参数 x 相当于形参
```

则语句：

```
cout << "面积：" << 3.14159 * r * r << endl;
```

可简写成：

```
cout << "面积：" << AREA(r) << endl;          // 使用时的参数 r 相当于实参
```

编译预处理时，编译器会将有参宏“AREA(r)”展开成“3.14159 * r * r”，展开时用实参 r 替换宏定义中的形参 x。

使用有参宏时的实参可以是表达式。例如“AREA(3+4)”，该有参宏在编译预处理时会被展开成“3.14159 * 3+4 * 3+4”。因为运算符优先级的问题，这个展开式是错误的。定义有参宏时使用小括号就可以解决此类问题，例如将上述有参宏 AREA(x)改写成如下形式：

```
#define   AREA(x)   3.14159 * (x) * (x)          // 用小括号将参数 x 括起来
```

此时“AREA(3+4)”会被展开成“3.14159 * (3+4) * (3+4)”，这个表达式是正确的。

有参宏可以带多个参数，例如：

```
#define   MAX(x, y)   ( ((x) > (y)) ? (x) : (y) )       // 求 x、y 中的较大值
#define   MIN(x, y)   ( ((x) < (y)) ? (x) : (y) )       // 求 x、y 中的较小值
```

#### 3. 空宏

**编译预处理指令语法：空宏定义指令**

| #define 宏名 |
|---|
| 语法说明：<br>■ 宏名需符合标识符的命名规则，无宏文本。 |

定义空宏不是用于宏替换，而是配合条件编译指令使用，详见 6.2.3 节。

#### 4. 宏删除

**编译预处理指令语法：宏删除指令**

| #undef 宏名 |
|---|
| 语法说明：<br>■ 宏名指定将要被删除的宏。 |

已定义的无参宏、有参宏或空宏都可以用宏删除指令（#undef）删除。删除后的宏不能继续使用，但还可以再次定义。

### 6.2.3　条件编译指令

程序开发过程中，源程序可能有多个版本，例如调试（debug）版本、发行（release）版本，或不同语种的版本等。如果用不同程序文件存放不同版本的源代码，文件数量会迅速增加，而且也很容易导致代码修改时的不一致问题。条件编译指令允许程序员将不同版本的源代码编写在同一程序文件中，便于管理和维护修改。

例 6-4 是一个计算圆面积和周长的程序，其 cout 语句显示的提示信息都是中文的（即中文界面），只有懂中文的用户才能使用这个程序。如何使用条件编译指令为该程序增加一个英文版本呢？让我们先来了解一下条件编译指令的语法。常用的条件编译指令有两种语法格式，格式 1 使用宏名，格式 2 使用常量表达式。

### 1. 条件编译指令（格式 1）

**编译预处理指令语法：条件编译指令(格式 1)**

```
#ifdef 宏名
    代码段 1
#else
    代码段 2
#endif
```

语法说明：

- 编译器在编译这段代码时，如果指定的**宏名**已定义，则编译**代码段** 1，否则编译**代码段** 2。这种格式的条件编译指令一般与空宏搭配使用。
- 如果没有代码段 2，则可以省略**#else**。

例 6-5 应用条件编译指令（格式 1）将例 6-4 的 C++程序改写成一个中英文混合版本，其中的代码第 5 行定义了一个空宏 ENGLISH_VERSION。编译连接该程序将生成一个英文版可执行程序。删除或注释掉代码第 5 行的空宏定义指令，重新编译连接则生成一个中文版可执行程序。

**例 6-5　应用条件编译指令的 C++演示程序（格式 1）**

```
1  #include <iostream>
2  using namespace std;
3
4  #define   PI   3.14159                 // 为 π 定义一个符号常量，这是一个无参宏
5  #define   ENGLISH_VERSION              // 定义一个空宏 ENGLISH_VERSION
6
7  int main( )                            // 主函数
8  {
9  #ifdef   ENGLISH_VERSION   // 若空宏 ENGLISH_VERSION 已定义，则编译以下 cout 语句
10     cout << "Input a radius please: " ;    // 显示英文提示信息
11 #else                                  // 否则，编译以下 cout 语句
12     cout << "请输入圆的半径：" ;           // 显示中文提示信息
13 #endif
14
15     double r;
16     cin >> r;     // 上述 2 条语句在中英文版中都需要，不用条件编译
17
18 #ifdef   ENGLISH_VERSION     // 若空宏 ENGLISH_VERSION 已定义，则编译以下代码段
19     cout << "Area: " << PI * r * r << endl ;
20     cout << "Perimeter: " << PI * 2 * r << endl ;
21 #else                           // 否则，编译以下代码段
```

```
22        cout << "面积：" << PI * r * r << endl ;
23        cout << "周长：" << PI * 2 * r << endl ;
24    #endif
25        return 0;                    // 中英文版都需要这条语句，不用条件编译
26    }
```

## 2. 条件编译指令（格式 2）

**编译预处理指令语法：条件编译指令(格式 2)**

```
#if 常量表达式
    代码段 1
#else
    代码段 2
#endif
```

语法说明：

- 编译器在编译这段代码时，如果指定的**常量表达式**结果不为 0，则编译**代码段** 1，否则编译**代码段** 2。常量表达式只能包含字面常量或符号常量。
- 如果没有代码段 2，则可以省略**#else**。

在保持程序功能不变的前提下，应用条件编译指令（格式 2）可以将例 6-5 修改为例 6-6 所示的程序。两者的主要区别就是代码第 5 行的宏定义指令。例 6-6 将原来的空宏 ENGLISH_VERSION 改为无参宏，即 ENGLISH_VERSION 是一个符号常量，其值为 1。这时代码第 9 和第 18 行条件编译指令中的 ENGLISH_VERSION 随之变成了一个常量表达式，编译器将根据该表达式结果决定编译哪个代码段。程序员将符号常量 ENGLISH_VERSION 的值设为 1（或任意非 0 值），编译连接将生成英文版可执行程序；设为 0，重新编译连接则生成中文版可执行程序。

**例 6-6　应用条件编译指令的 C++演示程序（格式 2）**

```
1     #include <iostream>
2     using namespace std;
3
4     #define  PI   3.14159               // 为 π 定义一个符号常量
5     #define  ENGLISH_VERSION  1         // 定义一个符号常量 ENGLISH_VERSION
6
7     int main( )                         // 主函数
8     {
9     #if  ENGLISH_VERSION // 若常量表达式 ENGLISH_VERSION 的值不为 0，则编译以下 cout 语句
10        cout << "Input a radius please: " ;   // 显示英文提示信息
11    #else                               // 否则，编译以下 cout 语句
12        cout << "请输入圆的半径：" ;          // 显示中文提示信息
13    #endif
14
15        double r;
16        cin >> r;                       // 上述 2 条语句在中英文版中都需要，不用条件编译
17
```

```
#if   ENGLISH_VERSION // 若常量表达式 ENGLISH_VERSION 的值不为 0，则编译以下代码段
      cout << "Area: " << PI * r * r << endl ;
      cout << "Perimeter: " << PI * 2 * r << endl ;
#else                                    // 否则，编译以下代码段
      cout << "面积：" << PI * r * r << endl ;
      cout << "周长：" << PI * 2 * r << endl ;
#endif
      return 0;                          // 中英文版都需要这条指令，不用条件编译
}
```

## 本节习题

1. 关于编译预处理指令，下列叙述正确的是（　　）。
   A. C++源程序中，一行可以编写多条编译预处理指令
   B. C++源程序中，编译预处理指令必须位于其他语句之前
   C. 宏替换不占用运行时间，只占编译时间
   D. 使用有参宏时，参数的类型必须与宏定义时一致
2. 在源程序中插入编译系统提供的头文件 iostream，下列语句中正确的是（　　）。
   A. #include iostream　　B. #include <iostream>;
   C. #include < iostream >　　D. #import <iostream>
3. 计算机执行下列 C++语句：

```
#define SUM(x, y)   x+y
cout << SUM(1, 2)*SUM(1, 2);
```

   执行结束后，显示器将显示（　　）。
   A. (1+2)*(1+2)　　B. 1+2*1+2　　C. 9　　D. 5
4. C++语言有三种形式的宏。下列哪种宏定义是错误的？（　　）
   A. #define ABC　　B. #define ABC{x}　x+10
   C. #define ABC　10　　D. #define ABC(x)　x+10
5. 编译如下的 C++代码：

```
#if   2
      代码段 1
#else
      代码段 2
#endif
```

   哪个代码段会被编译？（　　）
   A. 代码段 1
   B. 代码段 2
   C. 代码段 1 和代码段 2 都会被编译
   D. 代码段 1 和代码段 2 都不会被编译

# 6.3　几种特殊形式的函数

本节介绍 C++语言中 5 种特殊形式的函数，分别是带默认形参值的函数、重载函数、内联函数、带形参和返回值的主函数以及递归函数。

## 6.3.1　带默认形参值的函数

如果形参在多数情况下都取某个固定的值，程序员可以在定义函数或声明函数原型时将该值指定为形式参数的默认值，这就是**带默认形参值**的函数。调用带默认形参值的函数时，如果给出实参值，则将实参值赋值给形参变量，否则将默认值赋值给形参变量。

**问题举例：**编写一个兑换人民币的 C++程序，计算 500 元人民币分别可以兑换多少美元、欧元、英镑或港币。人民币汇率参见表 6-1。要求编写一个函数 Exchange 来实现兑换人民币的功能。

**表 6-1　人民币汇率表（2017 年 4 月 5 日）**

| | 可兑换人民币 |
|---|---|
| 1 美元 | 6.8993 |
| 1 欧元 | 7.3721 |
| 1 英镑 | 8.6119 |
| 1 港币 | 0.8878 |

**分析：**函数 Exchange 需定义两个形参，一个是兑换金额 amount，另一个是汇率 rate。函数返回值是兑换出的某币种金额。假设多数情况下调用函数 Exchange 都是为了兑换美元，那么可以将美元兑人民币的汇率 6.8993 设置成 rate 的默认值。这样在调用函数 Exchange 兑换美元时就可以省略汇率实参，简化调用语句。例 6-7 给出了兑换人民币的 C++程序。

**例 6-7　一个兑换人民币的 C++程序**

```
#include <iostream>
using namespace std;

double Exchange(double amount, double rate = 6.8993)
{
    return amount / rate;
}

int main( )
{
    cout << Exchange( 500 ) << endl;            // 将 500 元人民币兑换成美元，使用默认汇率
    cout << Exchange( 500, 7.3721 ) << endl;    // 将 500 元人民币兑换成欧元，给出汇率实参
    cout << Exchange( 500, 8.6119 ) << endl;    // 将 500 元人民币兑换成英镑，给出汇率实参
    cout << Exchange( 500, 0.8878 ) << endl;    // 将 500 元人民币兑换成港币，给出汇率实参
    return 0;
}
```

例 6-7 中主函数调用 Exchange 函数（第 11 行）计算 500 元人民币可以兑换多少美元时只给了一个金额实参 500，没有给汇率实参，此时将自动使用汇率默认值 6.8993 作为第二个实参。代码第 12~14 行分别兑换欧元、英镑和港币，调用 Exchange 函数都给出了汇率实参，此时汇率将以实际给出的实参为准。

带默认形参值函数的语法细则：

（1）**带默认值的形参**。调用时如果给出实参值，则将实参值赋值给形参变量；如果没有，则将默认值赋值给形参变量。不带默认值的形参在调用时必须给出实参，否则属于语法错误。

（2）**在函数原型声明中指定默认值**。如果函数定义在调用语句之后，应该在调用语句之前对函数原型进行声明。可以在声明语句中指定形式参数的默认值，此时函数定义中不能再指定默认值。函数具有文件作用域，同一函数在相同作用域中只能指定一次默认值。如果函数定义在其他文件中，应该在调用语句之前声明该函数的原型。可以在声明语句中指定形式参数在本文件中的默认值，并且可以与原函数定义中的默认值不同。

（3）**同一函数在不同作用域中可以指定不同的默认值**。如果多个默认值同时有效，调用函数时根据**局部优先原则**选择默认值。例 6-8 给出一个在不同作用域为函数形参指定不同默认值的 C++演示程序。

**例 6-8　一个在不同作用域为函数形参指定不同默认值的 C++演示程序**

```
1   #include <iostream>
2   using namespace std;
3
4   void fun(int p = 10);              // 指定文件作用域的形参默认值：10
5
6   int main( )
7   {
8       fun( );                        // 使用文件作用域的默认值，函数 fun 的显示结果：10
9       {                              // 为演示语法，此处刻意使用一条复合语句
10          void fun(int p = 20);      // 指定块作用域的形参默认值：20
11          fun( );          // 使用块作用域的默认值（局部优先），函数 fun 的显示结果：20
12      }
13      return 0;
14  }
15
16  void fun(int p)    // 因为第 4 行声明语句已指定了文件作用域的默认值，此处不能再指定
17  {
18      cout << p << endl;             // 显示形参 p 接收到的实参值
19  }
```

（4）**带默认值的形参必须定义在形参列表的后面**。形参列表中，可能有的形参带默认值，有的不带。定义函数或声明函数原型时，必须把带默认值的形参放在不带默认值形参的后面。例如：

```
void fun(int p1 = 10, int p2 = 20, int p3 = 30);        // 正确
void fun(int p1, int p2 = 20, int p3 = 30);             // 正确
void fun(int p1 = 10, int p2 = 20, int p3 );            // 语法错误
void fun(int p1 = 10, int p2, int p3 = 30);             // 语法错误
```

## 6.3.2　重载函数

函数名是函数的标识，调用函数时要通过函数名指定调用哪个函数。通常，同一文件中的函数之间不能重名，不同文件中的非静态函数（即外部函数）之间也不能重名。C++语言规定：如果两个函数形参的个数不同，或数据类型不同，那么这两个函数就可以重名。这样的重名函数被称为**重载函数**（overloaded function）。

将两个或两个以上的函数定义为重载函数的原因是这些函数的功能相同或相近，使用相同的名字便于程序员记忆，也不需要绞尽脑汁去想如何起不同的名字。例如下列 3 个求最大值的函数：

```
（1）int Max(int x, int y)   {   return ( (x > y) ? x : y );   }
（2）double Max(double x, double y)   {   return ( (x > y) ? x : y );   }
（3）int Max(int x, int y, int z)
    {
        int m;
        m = (x > y) ? x : y ;
        m = (m > z) ? m : z ;
        return m;
    }
```

这 3 个函数都是求最大值的，功能相同，但形参个数或类型不同。将这 3 个函数都命名为 Max，这就构成了重载函数。但这 3 个同名函数能够被正确调用吗，调用函数 Max 到底会调用哪一个呢？问题的答案是：编译源程序时，由编译器根据调用语句中实参的个数和类型自动调用形参最匹配的那个重载函数。例如下列 3 条函数调用语句：

```
cout << Max( 9, 5 ) ;          // 自动调用上面的函数(1)，即：int Max(int x, int y)
cout << Max( 9.0, 5.0 ) ;      // 自动调用上面的函数(2)，即：double Max(double x, double y)
cout << Max( 9, 5, 17 ) ;      // 自动调用上面的函数(3)，即：int Max(int x, int y, int z)
```

在应用重载函数时，如果两个函数仅仅是返回值类型不同，或形参名不同，那么程序员必须为它们起不同的名字。不能将这两个函数命名成重载函数，否则编译时将出现语法错误。另外，最好不要将两个功能差异很大的函数命名成重载函数，虽然没有语法错误，但会给程序员造成混淆。

## 6.3.3　内联函数

回顾一下计算机执行函数调用的具体过程：

（1）计算机执行到函数调用语句时将暂停主调函数的执行，跳转去执行被调函数。

（2）跳转前首先为被调函数定义的形参分配好内存，然后计算调用语句中的实参表达式，并将表达式结果按位置顺序一一赋值给对应的形参，即形实结合。

（3）保存好返回地址和当前 CPU 状态，即保存调用前的现场。

（4）跳转去执行被调函数的函数体。

（5）执行完被调函数的函数体或执行到其中的 return 语句时，退出被调函数的执行。如果有返回值，将返回值传回主调函数。

（6）恢复主调函数调用前的现场，调用结束。计算机按照返回地址继续执行主调函数中剩余的指令。

可以看出，为了实现函数跳转和数据传递，计算机需要执行一些额外的操作。实现相同的功能，单一主函数程序比主函数+子函数程序的执行速度要快，即函数跳转会降低程序的执行效率。但函数是团队分工协作和代码重用的基础，函数能提高程序的开发效率。

**内联函数**（inline function）是一种特殊的函数，它在保证程序开发效率的同时，不降低程序的执行效率。内联函数的原理是：编译源程序时将函数代码直接嵌入到每一个调用语句处，而在执行时不再进行函数跳转和数据传递。

内联函数的语法细则：

（1）**关键字 inline**。在函数定义的函数头前面加关键字 inline。如果函数定义在调用语句之后，或在其他文件中，则在调用前声明该函数原型时加关键字 inline。例 6-9 就是修改例 6-7 中的函数 Exchange，将其定义为内联函数。

**例 6-9　一个兑换人民币的 C++程序（内联函数）**

```
#include <iostream>
using namespace std;

inline double Exchange(double amount, double rate = 6.8993)        // 定义成内联函数
{
      return amount / rate;
}

int main( )
{
      cout << Exchange( 500 ) << endl;                // 将 500 元人民币兑换成美元，使用默认汇率
      cout << Exchange( 500, 7.3721 ) << endl;        // 将 500 元人民币兑换成欧元，给出汇率实参
      cout << Exchange( 500, 8.6119 ) << endl;        // 将 500 元人民币兑换成英镑，给出汇率实参
      cout << Exchange( 500, 0.8878 ) << endl;        // 将 500 元人民币兑换成港币，给出汇率实参
      return 0;
}
```

例 6-9 仅仅是在 Exchange 函数头的前面加上关键字 inline，将其定义成内联函数，而主函数中的调用语句保持不变。将函数改成内联函数不影响该函数原来的调用形式。

（2）**内联函数需是简单的函数**。编译器不能保证程序员所定义或声明的内联函数最终都能够按照内联的方式进行编译。如果该函数的函数体比较复杂（例如包含循环语句），编译器将自动按照非内联的方式进行编译。

（3）**内联函数的执行效率**。内联函数只有被多次调用（例如上万次），其执行效率才能体现出来，因此一般只是将频繁调用的简单函数定义成内联函数。

### 6.3.4　主函数 main 的形参和返回值

计算机系统包括硬件和软件两部分。**操作系统** OS（Operating System）是计算机系统中最基础、最重要的软件。操作系统直接运行于硬件之上。启动计算机后，操作系统被自动加载、执行。只有在操作系统运行之后，其他软件才能运行。

用户执行某个程序，实际上是给操作系统下达了一个执行程序的指令。在命令行界面操作系统中，若用户想执行某个名为 test.exe 的程序，则可以在命令行输入如下的指令：

test.exe　或　test

操作系统接收该指令，然后将程序文件 test.exe 读入内存，找到该程序的主函数 main，从主函数的第一条语句开始执行。程序执行结束后，从主函数返回操作系统。可以将操作系统执行某个程序的过程理解成操作系统调用该程序主函数 main 的过程。程序员编写主函数 main 时可以定义形参从操作系统接收数据，也可以向操作系统传递返回值（用于返回程序的运行状态）。

用户在向操作系统下达执行程序指令的时候，可以通过操作系统向程序传递某些原始数据。例如，用户向操作系统下达如下执行程序指令：

test　123 abc +++++

该指令在执行 test.exe 的同时将向该程序的主函数 main 传递 4 个字符串，它们依次是："test.exe""123""abc"和"+++++"，其中第 1 个字符串是该程序的文件名，后面 3 个是用户输入给程序的数据。也就是说，操作系统在调用程序主函数 main 时，可以将用户输入的数据以实参的形式传递给主函数。

在图形用户界面操作系统中，程序图标的背后实际上暗含了一个指向某个程序文件的链接。双击程序图标，操作系统将按照链接加载程序文件并执行该程序，其调用主函数 main 的过程与命令行界面操作系统是一样的。

#### 1. C++程序中的主函数 main

C++语言标准对主函数 main 有如下规定：

（1）一个 C++程序必须有并且只有一个名为 main 的主函数，主函数不能被重载。

（2）主函数是程序执行的起点。

（3）主函数的函数类型应为 int 型，需返回一个 int 型整数。

（4）主函数可以定义形参来接收实参数据，也可以省略形参（此时操作系统所传递过来的实参数据将被忽略）。

C++语法：主函数 main 的定义形式

| 有参形式 | 无参形式 |
| --- | --- |
| int main( int argc, char* argv[ ] )<br>{<br>    /*<br>    此处定义函数体代码<br>    */<br>    return 0;<br>} | int main( )<br>{<br>    /*<br>    此处定义函数体代码<br>    */<br>    return 0;<br>} |

**语法说明：**

- **argc** 表示所接收到的参数个数。
- **argv** 是一个 char 型指针数组，数组元素分别为 argv[0] ~ argv[argc-1]。参数以字符串形式传递，其中 argc[0]所指向的字符数组存放的是该程序的文件名，argc[1]所指向的字符数组存放的是第 1 个实参数据……
- 采用无参形式时，操作系统所传递过来的实参数据将被忽略。
- 主函数通过返回值将自己的运行状态返回给操作系统。通常用 0 表示正常，用-1 表示异常。

一个典型的带形参和返回值的主函数定义形式如例 6-10 所示。

例 6-10　典型的带形参和返回值的主函数定义形式

```
#include <iostream>
using namespace std;

int main( int argc, char *argv[ ] )
{
    for (int n = 0; n < argc; n++)          // 接收用户输入的参数：使用循环语句
        cout << argv[n] << endl;            // 本例演示如何显示这些参数

    // ……以下代码省略

    return 0;    // 返回值：任意整数值，通常用 0 或–1 来分别表示正常退出或异常退出
}
```

### 2. Microsoft C++编译器对主函数语法处理的差异

美国微软公司所开发的 C++编译器在对主函数语法的处理上存在某些差异，主要体现在 Visual C++ 6.0 和 Visual Studio 系列这两个集成开发环境的使用上。

**1）Visual C++ 6.0 集成开发环境**

Visual C++ 6.0 中的主函数 main 可以没有返回值，即函数类型可定义成 void。例如：

```
void main( )
{
    // 此处定义函数体代码
    return;                         // 如果 return 语句是最后一条语句则可以省略
}
```

**2）Visual Studio 系列集成开发环境**

程序员可以使用 Visual Studio 系列集成开发环境中的应用程序向导新建 Win32 控制台应用程序项目。假设新建一个名为 test 的项目，则应用程序向导将自动创建一个如下的 C++源程序文件：

```
// test.cpp : 定义控制台应用程序的入口点
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[ ])
{
   return 0;
}
```

该程序有两个主要的特征，一是插入了头文件 stdafx.h（其中包含一些常用的头文件），二是将主函数名改为_tmain，并且第二个形参的数据类型是_TCHAR *。之所以做这样的修改，是因为 Visual Studio 系列集成开发环境同时支持 ANSI 编码和 Unicode 编码的程序开发，改用_tmain 可以很方便地在这两种字符编码之间进行切换。初学者可直接将上述源程序文件改写成如下的形式：

```
#include "stdafx.h"                    // 保留该编译预处理指令
#include <iostream>
using namespace std;
int main( )                            // 修改主函数的定义
{
      // 此处定义函数体代码
      system( "pause" );               // 该语句的作用是暂停程序执行，以便程序员检查运行结果
      return 0;
}
```

也可以在新建项目时选择“空项目”，然后自己添加源程序文件，自己输入主函数 main 的定义代码（此时要去掉编译预处理指令：#include "stdafx.h"）。

另外，在使用 Visual C++ 6.0 或 Visual Studio 系列集成开发环境编写 Windows 图形用户界面程序（Win32 项目）时，程序员需将主函数名改为 WinMain 或_tWinMain，这是 Windows 图形用户界面程序执行时的起点。

## 6.3.5　递归函数

**问题举例**：编写一个求阶乘 $n!$的函数 Factorial，阶乘公式为

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

可以使用两种不同的方法来设计求阶乘算法，分别是递推法和递归法。

### 1. 递推法与递归法

**1）递推法**

递推法利用已知条件（0! = 1）和递推公式（$n! = n(n-1)!$），逐步递推求出 1!、2!、…，直到求出 $N$!。上述每个求解步骤都是已知$(n-1)!$求解 $n!$，可使用循环结构描述该算法，具体的函数代码如下。

```
int Factorial(int N)
{
    int result = 1;                        // 已知：0! = 1
    for (int n = 1; n <= N; n++)
        result = n * result;               // 递推公式：n! = n(n-1)!
    return result;
}
```

**递推法**求解问题的基本思想是：从**已知条件**出发，根据**递推公式**由简到繁，逐步逼近，最终求出问题的解，这种递推方法也称为**正向递推**。上述通过正向递推求解 $N$!问题时的每一步都是已知问题 $n-1$ 的解，递推求问题 $n$ 的解。这些递推步骤是在重复计算递推公式，可使用循环结构来描述递推算法。

**2）递归法**

递归法是程序设计中一种基于函数嵌套调用原理求解问题的方法。**递归法**求解问题的过程分两步完成：第一步是按照递推公式（此处称为**递归公式**）由繁到简，将求问题 $n$ 的解**降阶**成求问题 $n-1$ 的解，直到满足已知条件（称为**递归终结条件**）时返回已知结果，这个过程称为**逆向递推**；第二步是函数逐级返回结果，最终求出问题的解，这个过程称为**回归**。采用递归法求阶乘的函数代码如下：

```
int Factorial(int N)
{
    int result;
    if (N == 0)   // 递归终结条件：N == 0
        result = 1;   // 取得已知结果：0! = 1
    else
        result = N * Factorial( N-1 );   // 按照递归公式嵌套调用自身：参数降阶为 N–1
    return result;
}
```

该函数的函数体描述了求阶乘的递归过程，其中包含 3 个要素：

（1）递归终结条件。

（2）如果递归终结条件成立，则取得已知结果。

（3）如果递归终结条件不成立，则按照递归公式嵌套调用自身，即递归调用。

直接或间接调用自身的函数称为**递归函数**（recursive function）。与前面所讲述的带默认形参值的函数、重载函数或内联函数等函数不同的是，递归函数不仅仅是一个语法概念，其背后还暗含了递归法求解问题的算法设计思想。

### 2. 递归函数的定义与调用

C++语言使用递归函数描述递归算法。递归函数的定义或调用语法与普通函数没有什么两样，所不同的是递归函数的函数体应包含描述递归过程的 3 个要素，即**递归终结条件**、**已知结果**和**递归公式**。通常，一个典型的递归函数定义形式如下：

```
函数类型　函数名(形式参数列表)
{
    …
    if (递归终结条件)
        取得已知结果
    else
        按照递归公式调用自身
    …
}
```

例 6-11 给出了完整的递归法求阶乘 *N*!的 C++演示程序。

**例 6-11　递归法求解阶乘 N!的 C++演示程序**

```
#include <iostream>
using namespace std;

int Factorial(int N)
{
    int result;
    if (N == 0)                             // 递归终结条件：N == 0
        result = 1;                         // 取得已知结果：0! = 1
    else
        result = N * Factorial( N–1 );      // 按照递归公式嵌套调用自身：参数降阶为 N-1
    return result;
}

int main( )
{
    int x;
    x = Factorial( 3 );                     // 调用递归函数计算 3!
    cout << x << endl;
    return 0;
}
```

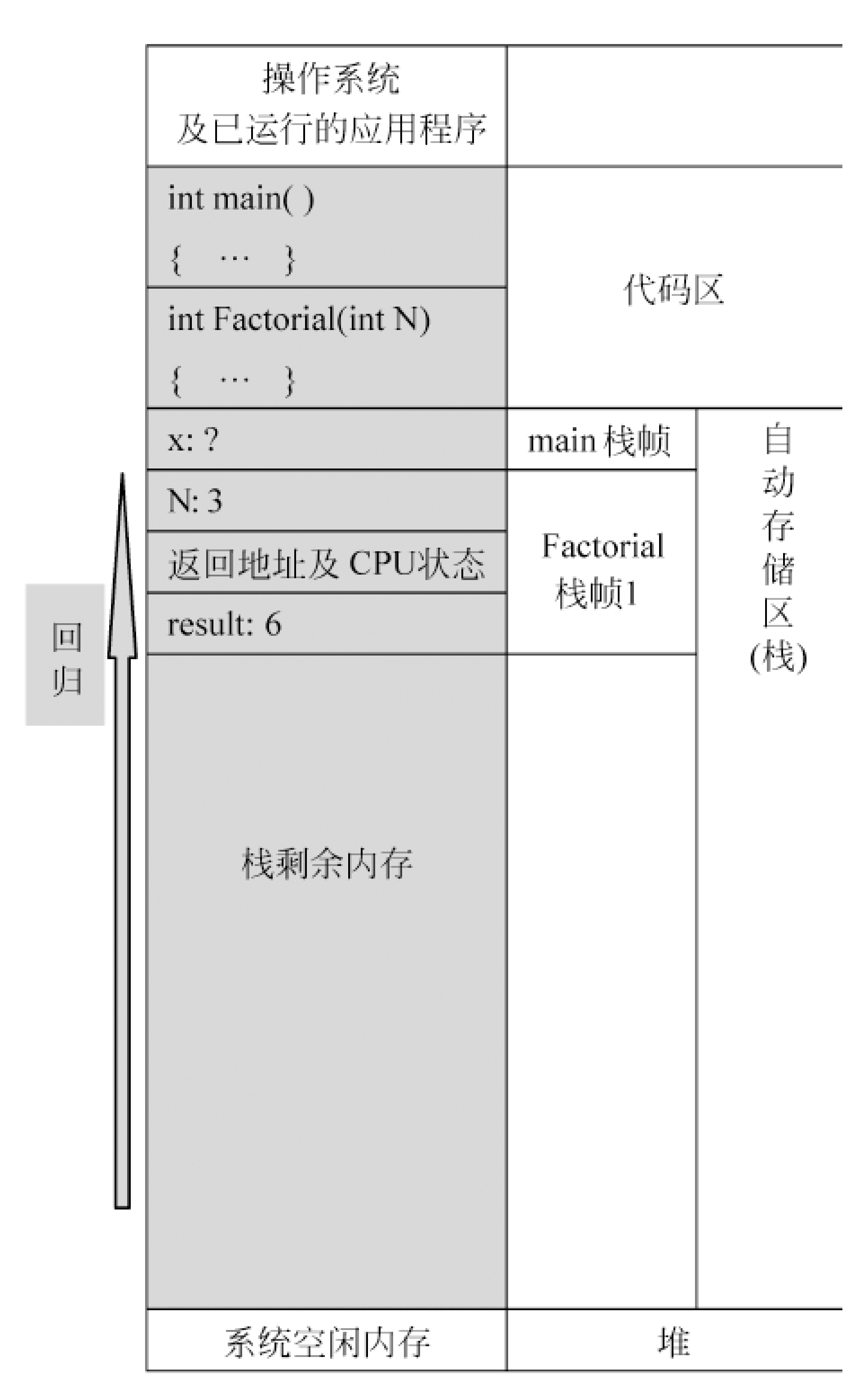
### 3. 递归函数的执行过程

计算机在执行函数调用语句跳转到被调函数时，为其形参及函数体中定义的局部变量分配内存，建立被调函数的栈帧。函数可以嵌套调用。每增加一级函数调用，栈帧就增加一个。每退出一级函数调用，栈帧就减少一个。计算机执行递归函数的过程就是递归函数不断嵌套调用自身、不断建立新栈帧的过程，即逆向递推的过程。当递归终结条件成立时停止嵌套调用，开始逐级返回结果、退出递归函数并依次释放栈帧，这就是回归的过程。

图 6-2 给出了执行例 6-11 中递归函数 Factorial 时的内存栈帧示意图。当执行主函数中调用函数 Factorial 的语句（代码第 17 行）时，计算机建立 Factorial 栈帧 1，形参 N 的值为 3。如果递归终结条件“N == 0”不成立，函数 Factorial 就不断嵌套调用自身（代码第 10 行）。每调用一次就新建一个栈帧，所不同的是 N 的值在逐级递减。在调用到第 4 次的时候，N 递减到 0，递归终结条件成立，此时停止调用，取得已知结果（0! = 1），直接将栈帧 4 中的 result 赋值为 1（代码第 8 行），此时的内存栈帧如图 6-2(a)所示。然后计算机开始逐级返回结果、退出递归函数并依次释放栈帧。图 6-2(b)是即将退出最上一级 Factorial 函数调用并准备返回主函数时的内存栈帧示意图，此时栈帧 1 中 result 所保存的数值就是 3!的计算结果 6。

操作系统
及已运行的应用程序
int main( )
{ … }
int Factorial(int N)
{ … }
代码区
x: ?
main栈帧
自动存储区(栈)
N: 3
返回地址及CPU状态
result: ?
Factorial 栈帧1
N: 2
返回地址及CPU状态
result: ?
Factorial 栈帧2
N: 1
返回地址及CPU状态
result: ?
Factorial 栈帧3
N: 0
返回地址及CPU状态
result: 1
Factorial 栈帧4
逆向递推
栈剩余内存
系统空闲内存
堆

(a) 第4次递归调用时的内存示意图




(b) 即将退出第1次递归调用时的内存示意图

图 6-2 例 6-11 递归函数 Factorial 的执行过程

递归函数的嵌套调用次数必须是有限的。无限调用将在执行时持续建立新栈帧，最终超出程序栈的内存范围，导致**栈溢出**（stack overflow）错误。因此编写递归函数需要注意以下两点：

（1）递归函数嵌套调用自身时所传递的实参一定要有变化，能够逐步向递归终结条件逼近。例如，例 6-11 代码第 10 行递归调用时的实参为 N−1，每次调用都将递减 1，逐步逼近递归终结条件“N == 0”。

（2）递归终结条件要考虑周全。例如，如果例 6-11 代码第 17 行的调用函数 Factorial 语句改为：

```
x = Factorial( –3 );                    // 假设主函数调用递归函数时给出了错误的实参，求–3!
```

此时，递归函数 Factorial 每次嵌套调用自身时递减 1，但永远无法逼近递归终结条件“N == 0”，最终导致栈溢出错误。将递归终结条件改为“N <= 0”就可以避免这样的错误。

### 4. 递归法与递推法的比较

针对某一具体的程序设计任务，选用递归法还是递推法，可考虑以下两方面的因素：

（1）**递归法比递推法速度慢**。递推算法用循环语句实现，执行速度快。递归算法用递归函数实现，但凡函数调用都需要一些额外的操作，因此执行速度相对较慢。

（2）**递归法比递推法适用范围广**。凡是递推法可以求解的问题都可以用递归法求解，反之则不然。某些问题很难用递推法求解，而用递归法则很简单。

**汉诺塔问题**：3 根塔针 A、B、C，A 上有 N 个盘子，大在下，小在上（见图 6-3）。要把 N 个盘子从 A 移到 C，每次只能移一个盘子。移动过程中可借助 B，但必须保持 3 根塔针上的盘子都是大在下，小在上。编程显示移动步骤。

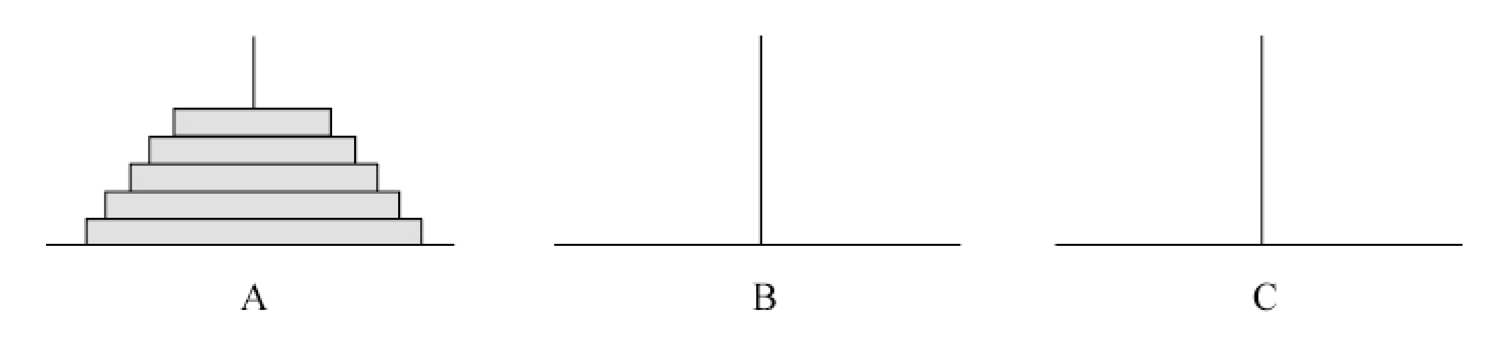



图 6-3　汉诺塔问题

**分析**：递推法很难求解汉诺塔问题，但使用递归法则很容易。算法思路如下：

（1）**1 个盘子**：直接从 A 移到 C，“N == 1”是递归终结条件。

（2）**N 个盘子**：把移动 N 个盘子的问题转化为移动 N–1 盘子的问题。

① 把 A 上 N–1 个盘子移到 B（借助 C）；

② 把第 N 个盘子从 A 移到 C；

③ 把 B 上 N–1 个盘子移到 C（借助 A）。

例 6-12 给出递归法求解汉诺塔问题的 C++程序。

**例 6-12　递归法求解汉诺塔问题的 C++程序**

```
#include <iostream>
using namespace std;

// 函数 hanoi 的功能：将 N 个盘子从 source 移到 destination（借助 relay）
void hanoi( int N, char source, char relay, char destination )          // 定义递归函数 hanoi
{
    if (N == 1)                          // 递归终结条件：N == 1
        // 如果只有一个盘子，则直接把这个盘子从 source 移到 destination
        cout << source << " --> " << destination << endl;
    else                                 // 否则进行递归
```

```
    {
        // 先把上面 N–1 个盘子从 source 移到 relay（借助 destination）
        hanoi( N–1, source, destination, relay );
        // 然后把第 N 个盘子从 source 移到 destination
        cout << source << " --> " << destination << endl;
        // 再把 relay 上的 N–1 个盘子移到 destination（借助 source）
        hanoi( N–1, relay, source, destination );
    }
}

int main( )
{
    cout << "The steps of moving 3 disks: " << endl;
    hanoi( 3, 'A', 'B', 'C' );        // 调用递归函数将 3 个盘子从 A 移到 C（借助 B）
    return 0;
}
```

例 6-12 的运行结果如下：

```
The steps of moving 3 disks:
A --> C
A --> B
C --> B
A --> C
B --> A
B --> C
A --> C
```

## 本节习题

1. 下列带默认形参值的函数定义中，语法错误的是（　　）。
   A. int fun(int x=0, double y=0.0) { … }　B. int fun(int x, double y=0) { … }
   C. int fun(int x=0, double y=1.5) { … }　D. int fun(int x=0, double y) { … }
2. 定义函数：int fun(int x=0) { … }，下列调用不正确的语句是（　　）。
   A. int y = fun(5);　B. int y = fun(0);
   C. int x = fun( );　D. int x = fun(5, 0);
3. 已有函数：void fun1(int x, double y) { … }，与该函数具有重载关系的是（　　）。
   A. int fun1(int x, double y) { … }　B. void fun2(double x, double y) { … }
   C. int fun2(int x, double y) { … }　D. void fun1(double x, double y) { … }
4. 已定义重载函数：void fun(int x) { … } 和 void fun(double x) { … }，下列哪种调用形式将调用第 2 个函数？（　　）
   A. fun( 5 );　B. fun( 5 + 3 );　C. fun( 5 – 3.0 );　D. fun( 5 / 3 );
5. 将函数定义为内联函数的目的是（　　）。
   A. 提高函数的执行效率　B. 将一个复杂程序划分成小的模块

C. 加强函数间的内部联系　　　　　　　D. 便于代码重用

6. 定义内联函数的关键字是（　　）。

A. 内联　　　　B. inline　　　　C. online　　　　D. include

7. 递归算法的基本要素是（　　）。

A. 形式参数和实际参数　　　　　　　B. 返回值和返回值类型

C. 终结条件、已知结果和递归公式　　D. 函数定义和调用

## 6.4 系统函数

程序员编写的函数可以在下一个项目中继续使用，即重用函数的代码。随着时间的推移，程序员将积累越来越多的函数，重用这些函数可以显著地提高开发效率。C++语言也预先编写了很多常用函数提供给广大程序员使用，这些函数被统称为**系统函数**。

C++语言是在 C 语言基础上发展而来的。C 语言是结构化程序设计语言，系统函数是其重要的附属组成部分。这些系统函数的源代码被编译成机器语言，以库文件（library，扩展名通常为.lib)的形式随编译系统提供。这些库文件被称为**标准 C 库**(Standard C Library)，通常存放在其安装目录下的 Lib 子目录中。调用标准 C 库中的系统函数都需要声明其函数原型。为方便程序员，C 语言预先编写好这些系统函数的原型声明语句，并按功能分类保存在若干个不同的头文件中。程序员只要用#include 指令包含相应的头文件，就可以调用这些系统函数了。程序连接时，被调用的系统函数代码将被连接到可执行程序文件中。

C++语言全盘继承了 C 语言的标准 C 库，另外又增加了一些新的库。这些新库被统称为 C++**标准库**（C++ Standard Library）。C++标准库又重写了一套标准 C 库中的系统函数，其目的是为这些函数增加 C++语言的类型安全检查和异常处理机制。另外，C++标准库还新增了一些新的系统函数，但更多的是新增了面向对象程序设计的**系统类库**（将在后续章节中陆续介绍）。C++标准库引入了命名空间的概念，所有程序实体，例如外部函数、全局变量或对象等，都定义在命名空间 std 中。

本节将首先介绍 C 语言的系统函数，然后再进一步介绍 C++语言中命名空间的概念以及 C++标准库。C++标准库的具体内容将在面向对象程序设计部分再做详细讲解。

### 6.4.1 C 语言的系统函数

标准 C 库提供了丰富的系统函数，其中包括输入/输出函数、数学函数、字符串处理函数和动态内存分配函数等。系统函数极大地扩展了 C 语言的功能，这使得程序员可以在更高的起点上开发程序。程序员在调用系统函数之前，需阅读编译系统提供的手册，学习各系统函数的功能及调用语法，并用#include 指令包含相应的头文件。标准 C 库头文件的扩展名都是“.h”。本节将简单介绍一些 C 语言中常用的系统函数。

#### 1. 输入/输出函数

C 语言本身并没有输入/输出语句，而是通过函数来实现数据的输入和输出。这里介绍

两个从键盘输入的标准输入函数（scanf、getchar），以及两个向显示器输出的标准输出函数（printf、putchar）。使用这四个函数需用#include 指令包含头文件<stdio.h>。

系统函数：格式化输入函数 scanf

int scanf( char *format, **变量地址列表** );

语法说明：

- 参数 **format** 是 char 型指针，接收一个格式控制字符串，其中包括格式符和分隔符。格式符是以"%"开头的字符串，用于指定输入数据的类型或格式（参见表 6-2）。
- **变量地址列表**指定保存输入数据的变量地址。一次可为多个变量输入数据，此时应为每个变量指定一个格式符，格式符应与变量的数据类型一致。多个变量地址之间用逗号","隔开。多个格式符之间通常用空格或逗号（即分隔符）隔开，输入数据时相应地也用空格或逗号分隔，以回车键结束。
- **返回值**是 int 型，返回输入数据的个数。
- 调用该函数时，计算机将暂停程序的执行，等待用户从键盘输入数据并以回车键结束。

举例：

```
int x;    scanf( "%d", &x );              // 输入十进制整数，保存到 int 型变量 x 中
float y;    scanf( "%f", &y );            // 输入十进制实数，保存到 float 型变量 y 中
double z;    scanf( "%lf", &z );          // 输入十进制实数，保存到 double 型变量 z 中
char ch;    scanf( "%c", &ch );           // 输入一个字符，保存到 char 型变量 ch 中
char str[20];    scanf( "%s", str );      // 输入一个字符串，保存到 char 型数组 str 中
scanf( "%d %f %lf %c %s", &x, &y, &z, &ch, str );     // 一次输入 5 个不同类型的数据
```

**表 6-2　常用格式符**

| | | |
|---|---|---|
| 整数 | %d | 十进制整数 |
| | %o | 八进制整数 |
| | %x | 十六进制整数 |
| 实数 | %f | float 型浮点数 |
| | %lf | double 型浮点数 |
| 字符 | %c | 单个字符 |
| | %s | 字符串 |

系统函数：格式化输出函数 printf

int printf( char *format, **表达式列表** );

语法说明：

- 参数 **format** 是 char 型指针，接收一个格式控制字符串，其中包括格式符和非格式符。格式符是以"%"开头的字符串，用于指定输出数据的类型或格式（参见表 6-2）。非格式符原样输出。

- printf 中的格式符可以指定输出数据的域宽（即显示时的占位宽度），实际数据达不到域宽时补空格。输出实数时还可以指定输出精度（即保留几位小数）。
- **表达式列表**指定需要输出的常量、变量或表达式。一次可输出多个表达式，此时应为每个表达式指定一个格式符，格式符应与表达式结果的数据类型一致。多个表达式之间用逗号“,”隔开。
- **返回值**是 int 型，返回输出数据的个数。
- 调用该函数时将按从右到左的顺序计算各表达式，然后按从左到右的顺序显示各表达式的结果。

举例：

```
int x=10;    printf( "x+5=%d", x+5 );          // 显示结果：x+5=15
float y=5.5;    printf( "y+1=%f", y+1 );        // 显示结果：y+1=6.5
double z=5.5;    printf( "z=%lf", z );          // 显示结果：z=5.5
char ch='A';    printf( "ch=%c", ch );          // 显示结果：ch=A
char str[20]="China";    printf( "%s", str );   // 显示结果：China
printf( "%5d, %5.2f, %5.2lf, %5c, %5s", x, y, z, ch, str );
// 一次显示多个表达式。格式符将每个数据的输出域宽都指定为 5，输出实数时保留 2 位小数
```

#### 系统函数：字符输入函数 getchar

```
int getchar( );
```

语法说明：

- 调用该函数时，计算机将暂停程序的执行，等待用户从键盘输入一个字符并以回车键结束。
- **返回值**是 int 型，返回所输入字符的 ASCII 码值。

举例：

```
char ch;    ch = getchar( );              // 输入一个字符，保存到变量 ch 中
```

#### 系统函数：字符输出函数 putchar

```
int putchar( int c );
```

语法说明：

- 调用该函数将变量 c 中的字符输出到显示器上。变量 c 中保存的是字符的 ASCII 码值。
- **返回值**是 int 型，返回所输出字符的 ASCII 码值。

举例：

```
putchar( 'A' );                          // 显示字符 A
putchar( 'A'+32 );                       // 显示字符 a
```

### 2. 数学函数

**头文件：**<math.h>
**常用函数：**平方根函数、指数函数、对数函数和各种三角函数等。

**举例**：例 6-13、例 6-14。

**例 6-13　求一元二次方程 ax ²+bx +c =0 根的 C 语言程序**

算法：如果 $\Delta = b^2 - 4ac \geqslant 0$，则根 $x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$；否则无实数根

```
#include <stdio.h>
#include <math.h>

int main( )
{
      double a, b, c;
      scanf( "%lf %lf %lf", &a, &b, &c );              // 从键盘输入 a, b, c 的值

      double d = b*b – 4*a*c;                          // 求 d（即Δ）
      if (d < 0)    printf( "无实数根\n" );
      else
      {
            double x1, x2;
            x1 = ( –b + sqrt(d) ) / (2*a);             // 函数 sqrt(d)：求 d 的平方根
            x2 = (–b – sqrt(d) ) / (2*a);
            printf( "x1=%lf, x2=%lf\n", x1, x2 );      // 显示实数根
      }
      return 0;
}
```

**例 6-14　画正弦函数 sin(x)波形的 C 语言程序（显示结果如图 6-4 所示）**

```
#include <stdio.h>
#include <math.h>
#define    PI    3.1415926
int main( )
{
      int n, angle, space;
      double val;
      // 画 0~360 度内 sin(x)的波形，每隔 18 度取一个采样点，一行显示一个采样点
      for (angle = 0; angle <= 360; angle += 18)
      {
            val = sin( angle*PI/180 );        // 函数 sin(x)：求角度 x 的正弦值，角度单位需转为弧度
            space = 20 + (int)( val*20 );    // 每行由若干个空格和一个星号组成，正弦值决定空格数
            for (n = 1; n <= space; n++)    putchar( ' ' );       // 每行先显示空格
            printf( "*\n" );       // 再显示星号“*”，并换行，准备显示下一个采样点
      }
      return 0;
}
```

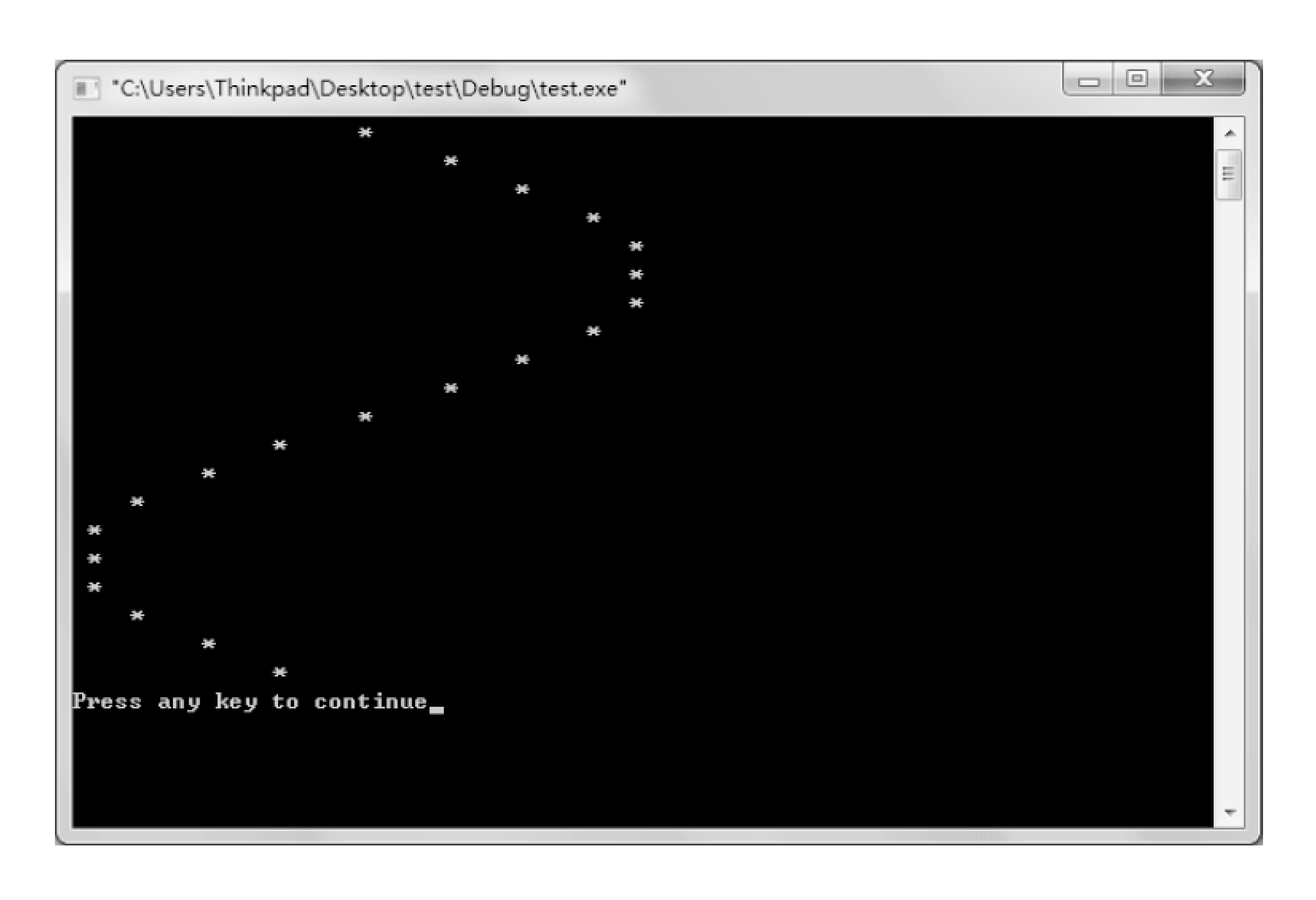



图 6-4　正弦函数 sin($x$)的波形

### 3. 字符串处理函数

**头文件：**<string.h>
**常用函数：**求字符串长度函数，字符串拷贝、连接等函数。
**举例：**例 6-15。

**例 6-15　演示字符串处理函数的 C 语言程序**

```c
#include <stdio.h>
#include <string.h>

int main( )
{
    char str1[20] = "Hello", str2[10];

    printf( "%d\n", strlen(str1) ); // 函数 strlen 求字符数组 str1 中字符串的长度，显示结果：5
    // 注 1：字符数组中字符串的长度指的是所保存的有效字符个数
    strcpy( str2, ", World!" );      // 函数 strcpy 将字符串", World!"复制到字符数组 str2 中
    // 注 2：已定义的字符数组不能直接赋值，例如：str2 = ", World!"; 这属于语法错误
    strcat( str1, str2 );            // 函数 strcat 将 str2 中的字符串连接到 str1 中字符串的后面
    printf( "%s\n", str1 );          // 显示连接后 str1 中的新字符串，显示结果：Hello, World!
    printf( "%d\n", strlen(str1) ); // 显示连接后 str1 中新字符串的长度，显示结果：13
    return 0;
}
```

### 4. 动态内存分配函数

**头文件：**<stdlib.h>
**常用函数：**分配内存函数、释放内存函数等。

**举例**：例 6-16。

**例 6-16　演示动态内存分配函数的 C 语言程序**

```
#include <stdio.h>
#include <stdlib.h>

int main( )
{
    int *p;

    p = (int *)malloc( 10*sizeof(int) );            // 动态分配一个含有 10 个元素的 int 型数组
    // 注：函数 malloc 返回所分配内存的首地址，其类型为“void *”，此处需转为“int *”

    for (int n = 0; n < 10; n++)                     // 使用循环结构遍历数组元素
    {
        *(p+n) = n*n;                                // 可以通过指针运算符访问第 n 个元素
        printf( "%d\n", p[n] );                      // 也可以通过下标运算符访问第 n 个元素
    }

    free( p );   // 释放指针变量 p 所指向的内存，即上面函数 malloc 动态分配的内存
    return 0;
}
```

上面简单介绍了 C 语言中几个常用系统函数的使用方法。喜欢深究的读者也许会问，这些系统函数是怎么实现的，例如标准输入函数 scanf 是怎么从键盘输入数据的。实话实说，本书的作者也不知道这些系统函数具体是怎么实现的。作者只能根据计算机硬件的基础知识来推测其实现原理。例如标准输入函数 scanf 应当是利用底层的中断服务来检测并接收键盘输入，经格式化转换后赋值给指定的变量。但不了解这些系统函数的内部细节并不影响程序员使用它们，正如不了解电视机的内部设计并不影响大家使用电视机。

程序员使用系统函数，首先要了解其功能，然后掌握其调用语法，其中包括函数名称、各形参的数据类型及其含义和返回值的数据类型及其含义。函数名、形式参数和函数类型（即返回值类型）共同组成了函数的调用接口。程序员调用系统函数实际上是重用其代码，实现相应的程序功能。

## 6.4.2　命名空间

编写程序时，程序员可以使用自己定义的函数，也可以使用系统函数，还可以使用任何从第三方购买的函数。C 语言和 C++语言都规定：所有的外部函数不能重名。但不同机构、不同程序员编写的函数难免会出现重名。

现实生活中也存在重名的问题，例如中国有一个“太原市”，越南也有一个“太原市”。单纯以“太原市”来看是重名的。但如果加上国度，例如“中国的太原市”和“越南的太原市”就不重名了。为地名加上国度可以有效地避免重名问题，这里的国度就是一种“命名空间”的概念。

C++语言引入了**命名空间**（namespace）的概念。不同机构或不同程序员可以将外部函

数和外部全局变量定义在各自的命名空间里，这样就能消除它们之间的重名问题。

### 1. 在命名空间中定义函数和全局变量

先使用关键字 namespace 定义一个命名空间，然后再将函数或全局变量定义在其后的一对大括号{ }中。例如，程序员 Tom 可以使用如下的语法形式为自己定义一个命名空间（假设也为 Tom），然后在命名空间中定义变量和函数。

```
namespace Tom                    // 定义一个命名空间 Tom
{
    int x, y;
    void fun1( )   {   …   }
    void fun2( )   {   …   }
}
```

上述代码在命名空间 Tom 中定义了两个全局变量 x、y，还定义了两个函数 fun1、fun2。

同一命名空间中的函数或全局变量之间不能重名，不同命名空间之间的函数或全局变量可以重名。

### 2. 访问命名空间中的函数和全局变量

访问命名空间中的函数和全局变量有 3 种方式。

**1）直接访问**

以“命名空间名::标识符”的形式直接访问，其中“::”为两个冒号（称为**作用域运算符**，或**作用域分辨符**）。例如：

```
Tom::x = 10;     Tom::y = 20;          // 访问命名空间 Tom 中的全局变量
Tom::fun1( );    Tom::fun2( );         // 调用命名空间 Tom 中的函数
```

**2）先声明，再访问**

先以“using 命名空间名::标识符;”的形式单独声明命名空间中的各标识符，然后再使用标识符访问。例如：

```
using Tom::x;                          // 先单独声明命名空间 Tom 中的各标识符
using Tom::y;
using Tom::fun1;
using Tom::fun2;
x = 10;     y = 20;                    // 然后再使用标识符访问
fun1( );    fun2( );
```

**3）统一声明**

先以“using namespace 命名空间名;”的形式统一声明命名空间里的所有标识符，然后使用标识符访问。例如：

```
using namespace Tom;                   // 先统一声明命名空间 Tom 里的所有标识符
x = 10;     y = 20;                    // 然后再使用标识符访问
fun1( );    fun2( );
```

using 和 namespace 都是 C++语言保留的关键字。C++语言还有一个默认的匿名命名空间。如果一个函数或全局变量未定义在任何命名空间里，则默认属于该匿名命名空间。

#### 3. 命名空间中外部函数和外部全局变量的声明

使用其他程序文件中定义的外部函数和外部全局变量，需要“先声明，再使用”。例如，如果命名空间 Tom 定义在其他程序文件中，则使用前需先声明其中的外部函数原型和外部全局变量，其声明形式如下：

```
namespace Tom                    // 声明命名空间 Tom 中的外部函数原型和外部全局变量
{
    extern   int x, y;           // 声明外部全局变量 x、y
    void fun1( );                // 声明外部函数 fun1 的原型
    void fun2( );                // 声明外部函数 fun2 的原型
}
```

### 6.4.3 C++语言的系统函数

C 语言曾是历史上使用最为广泛的程序设计语言。程序员也编写了大量的 C 语言程序，直到今天仍在使用。为了能够继续使用这些程序，同时也为了便于程序员平稳过渡到 C++语言，C++语言从设计之初就一直强调与 C 语言的兼容性。因此 C++语言全盘继承了 C 语言的语法规则，同时也全盘继承了 C 语言的标准 C 库。

在 C 语言结构化程序设计的基础上，C++语言又扩展了新的面向对象程序设计。C++标准库新增了一些系统函数，但更多的是新增了面向对象程序设计的系统类库。程序员编写 C++程序，可以继续使用原来的标准 C 库，也可以使用新的 C++标准库。

#### 1. 使用原标准 C 库

调用标准 C 库中的系统函数，需要用#include 指令包含相应的头文件。标准 C 库头文件共有 18 个，扩展名都是“.h”。C++程序可以按原来 C 语言的语法，继续包含原来的头文件，调用原来的系统函数。所有原来已经编写好的 C 语言程序在 C++编译器中也继续有效，可正常编译运行。

C++标准库又重写了一套标准 C 库中的系统函数，其目的是为这些函数增加 C++语言的类型安全检查和错误处理机制。C++标准库中新增的头文件采用了新的命名风格，去掉了扩展名“.h”，所有新增头文件都不再带扩展名。针对原来标准 C 库的 18 个头文件，C++标准库按新命名风格又另外重写了一套，去掉了“.h”扩展名，并在原文件名前加字母“c”。例如原头文件<stdio.h>，按新风格命名的头文件名为<cstdio>。新旧两套头文件的语法作用和功能完全相同，建议 C++程序员使用新的头文件。

#### 2. 使用 C++标准库中新增的系统函数

C++标准库新增了一些系统函数，所有新增的系统函数都定义在命名空间 std 中。调用这些函数除了使用#include 指令包含相应的头文件之外，还需要声明其命名空间。例 6-17 给出一个调用新增系统函数 swap 的 C++演示程序。

**例 6-17　调用新增系统函数 swap 的 C++演示程序**

```
#include <iostream>   // 包含新的头文件：iostream
using namespace std;   // 声明其命名空间：std

int main( )
{
    int x = 5, y = 10;

    cout << "x=" << x << ", y=" << y << endl;   // 显示结果：x=5, y=10
    swap(x, y);   // 调用函数 swap，交换变量 x、y 的值
    cout << "x=" << x << ", y=" << y << endl;   // 显示结果：x=10, y=5
    return 0;
}
```

### 3. 初步认识 C++标准库中的系统类库

C++是面向对象的程序设计语言。C++标准库更多的是基于面向对象程序设计方法所新增的系统类库。之前经常使用的 cin、cout 输入/输出指令实际上就是 C++标准库预先定义好的对象。cin 是一个标准输入流对象，而 cout 则是一个标准输出流对象。虽然可以继续使用标准 C 库里的输入/输出函数，但 C++程序员应当使用面向对象程序所提供的 cin、cout 对象来输入/输出数据。使用 cin、cout 对象要用#include 指令包含相应的头文件<iostream>，另外还需要对命名空间 std 进行声明。例如下面就是 C++程序经常使用的两条语句：

```
#include <iostream>
using namespace std;
```

C++语言可以在很大程度上替代 C 语言，例如用 cin、cout 对象代替了原来的 scanf、printf 函数，用字符串类 string 代替了原来的字符数组和字符串处理函数。这些内容将在后续面向对象程序设计的章节中再做详细讲解。另外，在面向对象程序设计中动态内存分配的作用更加重要。C++语言直接新增了 new 和 delete 这两个运算符，来取代原标准 C 库里的动态内存分配函数。

本节最后再简单描述一下多文件结构下程序员与函数的关系。

**1）为别人提供函数的程序员**

（1）将常用的功能或算法定义成函数，保存到源程序文件中（扩展名为.cpp）。将源程序文件编译成目标代码文件（扩展名为.obj），通常还会进一步将目标代码打包成函数库文件（扩展名通常为.lib）。

（2）为函数库中定义的函数编写声明语句，集中保存在一个头文件中（扩展名为.h）。

（3）发布或销售函数库产品，其中包含库文件和头文件。库文件为目标代码（即机器语言），其他程序员只能调用，无法阅读或修改。而头文件是函数声明语句的源代码，其他程序员可以阅读，以了解函数的功能与调用接口。

**2）使用别人函数的程序员**

（1）可以使用系统函数，也可以使用经合法渠道获得（比如购买）的任何第三方开发

的函数库。

（2）编写程序时可以调用别人函数库中的函数，从而实现某种特定的功能。调用前需要用#include 指令包含相应的头文件。

（3）连接时，连接器将函数库中被调函数的代码与程序员自己编写的代码连接到一起，形成最终的可执行程序。

### 本节习题

1. 定义 3 个变量“int x, y;　double z;”，使用标准 C 库中的格式化输入函数 scanf 输入 x、y 和 z 的值，下列输入语句中正确的是（　　）。

   A. scanf(x, y, z);　　B. scanf("%d%d%lf", x, y, z);

   C. scanf("%d%d%lf", &x, &y, &z);　　D. scanf("%int%int%double", x, y, z);

2. 执行 C++语句“int a = 21, b = 1;　printf("%2d%2d", a, b);”，显示器上将显示（　　）。

   A. 211　　B. 21 1　　C. 22121　　D. %2d%2d211

3. 执行 C++语句“int x = strlen("China 中国");”，执行后变量 x 的值为（　　）。

   A. 7　　B. 8　　C. 9　　D. 10

4. 关于命名空间，下列描述不正确的是（　　）。

   A. C++命名空间是用于解决命名冲突问题的

   B. 定义命名空间使用关键字 namespace

   C. 声明命名空间使用关键字 include

   D. 声明命名空间使用关键字 using

5. 关于 C++语言的系统函数，下列描述不正确的是（　　）。

   A. 使用 C++标准库中的系统函数需包含其相应的头文件

   B. C++标准库定义了一个命名空间 std

   C. C++标准库是 C++语言的重要扩展

   D. 编写 C++程序不能使用 C 语言中的系统函数

## 6.5 自定义数据类型

C++语言提供了比较完善的基本数据类型，其中包括整型（int, short, long）、浮点型（float, double）、字符型（char）和布尔型（bool）等 4 大类。程序员可以根据需要为这些基本数据类型重新命名一个别名，或基于这些基本数据来定义新的复杂数据类型，这些类型被统称为**自定义数据类型**。例如：

```
typedef  float  REAL;          // 为 float 类型重新命名一个别名 REAL（实数）
typedef  char   NAME[10];      // 基于 char 定义字符型数组，称为 NAME（姓名）类型
```

程序员可以将上述自定义数据类型当作一种新的数据类型来定义变量。

本节将介绍“typedef”类型定义语法，以及枚举、联合体、结构体等常用自定义数据类型的定义和使用方法。

## 6.5.1　类型定义 typedef

可以使用关键字 **typedef** 为基本数据类型重新命名一个别名，或为指针、数组等定义新的数据类型。

### 1. 为基本数据类型重新命名一个别名

例如，定义保存年龄的变量，有的程序员可能使用 int 型，有的使用 short 型或 unsigned char 型。一个程序开发项目需要统一数据类型的使用，这时可使用类型别名先定义一个年龄类型 AGE：

```
typedef   unsigned char   AGE;            // 年龄类型“AGE”等价于“unsigned char”
```

然后，程序开发项目要求所有程序员都必须使用类型别名 AGE 来定义保存年龄的变量，这样就可以将所有保存年龄变量的数据类型都统一为 unsigned char。例如：

```
AGE x;            // 等价于：unsigned char x;
```

### 2. 定义指针类型

可以为指针定义新的数据类型，这样可以简化程序代码。例如：

```
typedef   int *   IPointer;               // 定义一个 int 型指针类型 IPointer
typedef   double *   DPointer;            // 定义一个 double 型指针类型 DPointer
```

使用上述新数据类型定义指针变量：

```
IPointer   p1;                            // 等价于：int *p1;
DPointer   p2;                            // 等价于：double *p2;
```

### 3. 定义数组类型

可以为数组定义新的数据类型。例如：

```
typedef   char   NAME[9];                 // 定义一个字符数组类型，命名为 NAME
                                          // 使用 NAME 类型可定义专门保存姓名的变量
typedef   int   IArray[20];               // 定义一个 int 型数组类型，命名为 IArray
```

使用上述新数据类型定义数组变量：

```
NAME   name1, name2;                      // 等价于：char name1[9], name2[9];
IArray   a, b;                            // 等价于：int a[20], b[20];
```

实际上 C++语言本身也会使用 typedef 定义新数据类型，例如基于 UTF16 编码的宽字符类型 wchar_t 就是这样定义的：

```
typedef   unsigned short   wchar_t;       // 宽字符类型 wchar_t 实际上就是 unsigned short
```

将自定义数据类型与编译预处理指令搭配使用，可以方便跨平台移植程序，或统一管理多个版本的源代码。例 6-18 给出一个统一 ANSI 和 Unicode 这两种不同编码的 C++演示程序。

**例 6-18　统一 ANSI 和 Unicode 这两种不同编码的 C++演示程序**

```
1   #include <iostream>
2   using namespace std;
3
4   #define   _UNICODE              // 定义一个空宏_UNICODE
5   #ifdef _UNICODE
6       typedef   wchar_t   TCHAR;
7   #else
8       typedef   char   TCHAR;
9   #endif
10
11  int main( )
12  {
13      TCHAR   ch;                 // 编译时，该语句等价于：wchar_t ch; 即 Unicode 版本
14      // 如果删除第 4 行空宏_UNICODE 定义再编译，则该语句等价于：char ch; 即 ANSI 版本
15
16      // ……以下代码省略
17  }
```

## 6.5.2　枚举类型

和 C++语言中其他基本数据类型相比，布尔（bool）类型的主要特点是其值域只有两个取值，即真和假，分别用关键字 true 和 false 表示。实际程序设计任务也会经常碰到与布尔类型相似的数据，它们的值域是有限的（称为是可枚举的）。例如一个星期只有星期一、星期二……星期日 7 天，其值域是可枚举的。C++语言可以将值域可枚举的数据定义成新的数据类型，这些数据类型被统称为**枚举类型**，值域中的每个取值称为一个**枚举元素**。

### 1. 定义枚举类型

**C++语法：定义枚举类型**

```
enum 枚举类型名  { 枚举常量 1, 枚举常量 2, …, 枚举常量 n };
```

语法说明：

- **enum** 是定义枚举类型的关键字。
- **枚举类型名**需符合标识符的命名规则。
- **枚举常量**是表示各个枚举元素的名称，需符合标识符的命名规则。
- 计算机内部存储枚举型数据时，用整数表示各个枚举常量。默认情况下，枚举常量 1 = 0，枚举常量 2 = 1……枚举常量 n = n−1。可以在定义时为枚举常量另行指定其他的值。

举例：定义一个枚举类型 WeekDay

```
enum   WeekDay   { sun, mon, tue, wed, thu, fri, sat };        // 默认：sun=0, mon=1, …
```

或

```
enum   WeekDay   { sun=7, mon=1, tue, wed, thu, fri, sat };   // sun=7, mon=1, tue=2, …
```

### 2. 定义和访问枚举类型的变量

使用枚举类型可以定义变量，定义时需加 enum 关键字。定义好的枚举变量可以访问，即读或写。例 6-19 给出一个枚举类型的 C++演示程序。

**例 6-19　枚举类型的 C++演示程序**

```
#include <iostream>
using namespace std;

enum   WeekDay   { sun, mon, tue, wed, thu, fri, sat };   // 默认：sun=0, mon=1, …

int main( )
{
     enum   WeekDay   x;          // 定义一个 WeekDay 类型的枚举变量 x
     x = mon;   // 为 x 赋值，mon 是枚举常量，内部数值是 1
                    // 注：枚举变量不能直接用整数赋值。例如 x = 1; 是错误的，两者类型不一致
     cout << x << endl;              // 显示枚举类型数据将显示其对应的数值，显示结果：1
     return 0;
}
```

枚举类型便于程序员记忆，所编写的源代码也更容易理解。枚举类型可以进行关系运算，比较大小。比较枚举类型数据的大小实际上是比较其对应数值的大小。例如：

mon > sun　的结果为 true，因为 1 > 0
mon > tue　的结果为 false，因为 1 < 2

### 3. 使用 typedef 命名枚举类型

可以用 typedef 为枚举类型命名，这样在定义变量时就不再需要 enum 关键字，从而简化了程序代码。例如命名一个枚举类型 WEEKDAY：

```
typedef   enum { sun, mon, tue, wed, thu, fri, sat }   WEEKDAY;
WEEKDAY   x, y;                    // 用 WEEKDAY 类型定义两个枚举变量 x, y
```

## 6.5.3　联合体类型

早期的计算机内存比较小（例如 32KB 或 64KB），因此内存资源比较紧张。联合体类型是 C++语言为让同一程序中的多个变量共用内存，减少内存占用而专门设计的一种数据类型，例如下面给出的程序例子。

假设函数 fun 的函数体有 3 段代码，第 1 段代码需要定义一个 char 型变量 ch，第 2 段需要定义一个 int 型变量 x，第 3 段需要定义一个 double 型变量 y，其示意代码如下。

```
void fun( )
{
     char ch;                          // 第 1 段代码定义一个 char 型变量 ch
     ch = 'A';
     //…
```

```
    int x;                         // 第 2 段代码定义一个 int 型变量 x
    x = 10;
    //…

    double y;                      // 第 3 段代码定义一个 double 型变量 y
    y = 35.2;
    //…
}
```

在这个例子中，变量 ch、x 和 y 分别在 3 个不同的代码段内使用，时间上不会重叠。这时可以将这三个变量合并在一起定义成一种联合体类型，让它们共用同一内存单元。

### 1. 联合体类型的定义与使用

**C++语法：定义联合体类型**

```
union 联合体类型名
{
    数据类型 1   变量成员名 1;
    数据类型 2   变量成员名 2;
    …
    数据类型 n   变量成员名 n;
};
```

语法说明：

- **union** 是定义联合体类型的关键字。
- **联合体类型名和各变量成员名**需符合标识符的命名规则。
- 各变量成员之间的数据类型可以相同，也可以不同，但变量名不能相同。

联合体类型中包含多个变量成员，变量成员的数据类型可以各不相同。使用联合体类型可以定义变量，定义时需加 union 关键字。和基本数据类型的变量相比，联合体变量是一种复杂变量，其中包含多个下级成员，每个成员都相当于是一个变量。

访问联合体变量下级成员的语法形式是“**联合体变量名.下级成员名**”，其中圆点“.”称为**成员运算符**。使用联合体类型可以对之前程序例子中函数 fun 的代码做如下改写：

```
union UType                        // 定义一个联合体类型 UType，包含 ch、x 和 y 3 个成员
{
    char ch;
    int x;
    double y;
};
void fun( )
{
    union UType   a;               // 定义一个 UType 类型的联合体变量 a
    a.ch = 'A';                    // 访问联合体变量 a 的下级成员 ch
    // …

    a.x = 10;                      // 访问联合体变量 a 的下级成员 x
```

```
    // …
    a.y = 35.2;                    // 访问联合体变量 a 的下级成员 y
    // …
}
```

定义 UType 类型的联合体变量 a，计算机将为其分配内存空间（见图 6-5）。

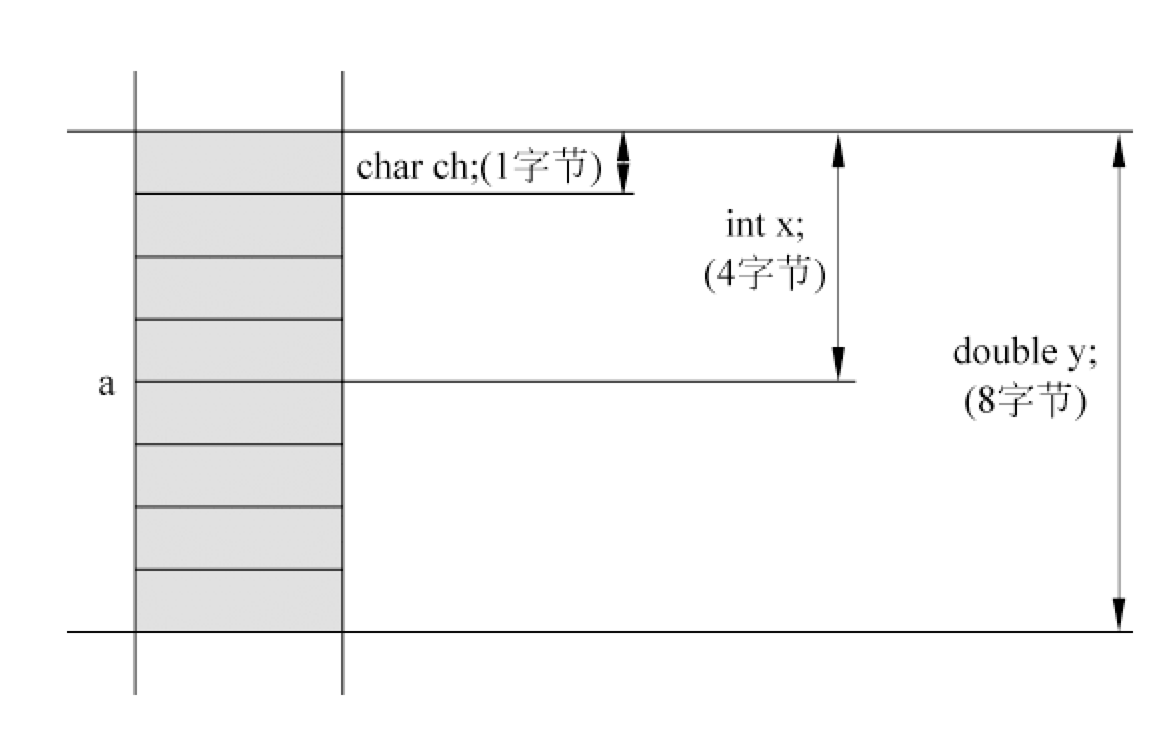



图 6-5　UType 类型联合体变量 a 的内存示意图

从图 6-5 可以看出，联合体变量成员之间是共用内存单元的，因此联合体类型也称为**共用体类型**。一个联合体变量所占用的字节数等于其最大成员的字节数。上例中，联合体变量 a 所占用的字节数等于其最大成员 y 的字节数(即 8 字节)。而在采用联合体类型之前，变量 ch、x 和 y 这 3 个变量共占用 13 个字节。

使用联合体类型需要注意的是，因为共用内存单元，联合体变量在同一时刻只能保存一个成员的数据。程序员应当准确理解联合体类型的这一特点，否则会造成数据丢失。

### 2. 使用 typedef 命名联合体类型

可以用 typedef 为联合体类型命名，这样在定义变量时就不再需要 union 关键字，从而简化了程序代码。例如：

```
typedef union
{
    char ch;
    int x;
    double y;
}  UTYPE;               // 命名一个联合体类型 UTYPE，包含 ch、x 和 y  等 3 个成员
UTYPE   a, b;           // 用 UTYPE 类型定义两个联合体变量 a, b
```

## 6.5.4　结构体类型

一个 C++程序可能需要处理大量的数据，定义很多个变量。如果按照某种关联关系对这些变量进行分类管理，那就能够降低管理难度。

例如一个模拟学校人员信息管理的 C++程序，其中包括学生信息和教师信息两部分。为简单起见，假定学生信息只包含学号、姓名、年龄和学分这 4 项，教师信息只包含教师编号、姓名、年龄和工资这 4 项。模拟学校人员信息管理 C++程序的示意代码如下。

```
int main( )
{
    // 定义保存第 1 位同学信息的变量
    char sID1[11], sName1[9];                 // 保存学号和姓名的变量
    int sAge1;                                // 保存年龄的变量
    double sScore1;                           // 保存学分的变量
    cin >>sID1 >> sName1 >> sAge1 >> sScore1;
    // 定义保存第 2 位同学信息的变量
    char sID2[11], sName2[9];                 // 保存学号和姓名的变量
    int sAge2;                                // 保存年龄的变量
    double sScore2;                           // 保存学分的变量
    cin >>sID2 >> sName2 >> sAge2 >> sScore2;

    // 定义保存第 1 位教师信息的变量
    char tID1[7], tName1[9];                  // 保存编号和姓名的变量
    int tAge1;                                // 保存年龄的变量
    double tSalary1;                          // 保存工资的变量
    cin >>tID1 >> tName1 >> tAge1 >> tSalary1;
    // 定义保存第 2 位教师信息的变量
    char tID2[7], tName2[9];                  // 保存编号和姓名的变量
    int tAge2;                                // 保存年龄的变量
    double tSalary2;                          // 保存工资的变量
    cin >>tID2 >> tName2 >> tAge2 >> tSalary2;
    // 后续处理代码省略
}
```

在这段示意代码中，为处理两位同学和两位教师的信息，程序总共需要定义 16 个变量来保存相关数据。因为各变量之间不能重名，所以程序员定义变量时分别采用前缀（s 或 t）和后缀（1 或 2）来区分各变量名。可以看出，在数据量较大时程序需要定义很多的变量，这时连为变量命名也是一件麻烦事。可以采用 C++语言提供的结构体类型对程序中的变量进行分类管理，其步骤分为两步：第一步先定义结构体类型，第二步再定义结构体类型的变量，然后访问结构体变量。下面采用结构体类型来改写前面的模拟学校人员信息管理 C++程序。

（1）定义结构体类型。

```
struct Student                     // 定义一个学生信息结构体类型 Student
{
    char ID[11], Name[9];          // 保存学号和姓名的变量
    int Age;                       // 保存年龄的变量
    double Score;                  // 保存学分的变量
};
struct Teacher                     // 定义一个教师信息结构体类型 Teacher
{
    char ID[7], Name[9];           // 保存教师编号和姓名的变量
```

```
        int Age;                        // 保存年龄的变量
        double Salary;                  // 保存工资的变量
};
```

与学生相关的学号、姓名、年龄和学分，它们都是学生信息的一部分，本来就是一个逻辑上的整体。上述示意代码将这 4 个与学生相关的信息划分成一类，定义一个结构体类型 Student。将保存学号、姓名、年龄和学分的变量（ID、Name、Age 和 Score）定义成该结构体类型的下属成员。同样地，将与教师相关的教师编号、姓名、年龄和工资等信息划分成一类，定义一个结构体类型 Teacher。

（2）在主函数中定义结构体类型的变量，然后访问结构体变量。

```
int main( )
{
        // 定义保存第 1 位同学信息的结构体变量 s1
        struct Student   s1;
        cin >> s1.ID >> s1.Name >> s1.Age >> s1.Score;
        // 定义保存第 2 位同学信息的结构体变量 s2
        struct Student   s2;
        cin >> s2.ID >> s2.Name >> s2.Age >> s2.Score;

        // 定义保存第 1 位教师信息的结构体变量 t1
        struct Teacher   t1;
        cin >> t1.ID >> t1.Name >> t1.Age >> t1.Salary;
        // 定义保存第 2 位教师信息的结构体变量 t2
        struct Teacher   t2;
        cin >> t2.ID >> t2.Name >> t2.Age >> t2.Salary;
        // 后续处理代码省略
}
```

定义好的结构体类型将被当作一种新的数据类型来定义变量，定义时需加 struct 关键字。例如，主函数用结构体类型 Student 定义了一个变量 s1，用于保存第 1 位同学的信息。

```
struct Student   s1;                          // 用 Student 类型定义一个结构体变量 s1
```

按照 Student 类型的定义，结构体变量 s1 中应包含 4 个成员，它们分别是 s1.ID（学号）、s1.Name（姓名）、s1.Age（年龄）和 s1.Score（学分）。结构体变量中的每个成员都是一个变量，可以单独访问它们。例如：

```
cin >> s1.ID >> s1.Name >> s1.Age >> s1.Score;   // 输入第 1 位同学的信息
```

可以看出，定义一个结构体变量相当于一次定义了多个普通变量。为了处理两位同学和两位教师的信息，采用结构体类型之前的程序定义了 16 个普通变量。而采用结构体类型之后，程序只定义了 4 个结构体变量，其中 s1 和 s2 用于保存两位同学的信息，t1 和 t2 用于保存两位教师的信息。采用结构体类型之后，程序条理更加清楚，可读性也提高了。

### 1. 定义结构体类型

结构体类型将多个具有内在关联关系的变量组合在一起形成一个逻辑上的整体，这些

变量成为整体的下属成员。结构体类型是一种由程序员定义出的复杂数据类型，其中可以包含多个变量成员。定义结构体类型就是声明其中包含了哪些变量成员，以及这些变量成员的数据类型。

**C++语法：定义结构体类型**

```
struct 结构体类型名
{
    数据类型 1   变量成员名 1;
    数据类型 2   变量成员名 2;
    …
    数据类型 n   变量成员名 n;
};
```

语法说明：

- **struct** 是定义结构体类型的关键字。
- **结构体类型名和各变量成员名**需符合标识符的命名规则。
- 各变量成员之间的数据类型可以相同，也可以不同，但变量名不能相同。

### 2. 定义和访问结构体类型的变量

定义好的结构体类型将被当作一种新的数据类型来定义变量，定义时需加 struct 关键字。所定义出的结构体变量是一种复杂变量，其中将包含多个下级成员。访问结构体变量，通常是访问其下级成员。

（1）同一结构体类型可以定义出多个变量，它们将包含相同的下级成员。例如：

```
struct Student   s1, s2;                  // 用 Student 类型定义两个结构体变量 s1 和 s2
```

s1 和 s2 都是用 Student 类型定义出来的，都属于 Student 类型的结构体变量。按照 Student 类型的定义，结构体变量 s1 和 s2 都包含 ID（学号）、Name（姓名）、Age（年龄）和 Score（学分）这 4 个下级成员。每个 Student 类型的结构体变量都可以保存一位同学的信息。用 Student 类型定义多少个结构体变量，就可以保存多少位同学的信息。

（2）不同结构体类型所定义出的结构体变量将包含不同的下级成员。例如：

```
struct Student   s;                       // 用 Student 类型定义一个结构体变量 s
struct Teacher   t;                       // 用 Teacher 类型定义一个结构体变量 t
```

结构体类型 Student 和 Teacher 是两个不同的数据类型，它们包含不同的下属成员。按照 Student 类型的定义，结构体变量 s 中将包含 s.ID（s 的学号）、s.Name（s 的姓名）、s.Age（s 的年龄）和 s.Score（s 的学分）这 4 个成员。而按照 Teacher 类型的定义，结构体变量 t 中将包含 t.ID（t 的教师编号）、t.Name（t 的姓名）、t.Age（t 的年龄）和 t.Salary（t 的工资）这 4 个成员。如需保存学生信息，那么就定义 Student 类型的结构体变量；如需保存教师信息，那么就定义 Teacher 类型的结构体变量。

（3）结构体变量的内存分配。和基本数据类型的变量一样，计算机在执行结构体变量

定义语句时，将为结构体变量分配内存空间。计算机会为结构体变量中的所有成员同时分配内存。理论上，每个结构体变量所占用的字节数等于其所有成员占用字节数的总和。图 6-6 给出了结构体变量 s（Student 类型）和 t（Teacher 类型）的内存分配示意图，其中 s 占 32 个字节，t 占 28 个字节。**注：**在实际应用中，为了提高结构体变量的内存访问速度，C++编译器会采用“字节倍数对齐”技术，每个结构体变量所占用的内存可能大于其成员占用字节数的总和。

| | | 各成员在结构体类型中的定义 |
|---|---|---|
| 结构体变量 s (Student类型) | 11 字节 | char ID[11]; |
| | 9 字节 | char Name[9]; |
| | 4 字节 | int Age; |
| | 8 字节 | double Score; |
| 结构体变量 t (Teacher类型) | 7 字节 | char ID[7]; |
| | 9 字节 | char Name[9]; |
| | 4 字节 | int Age; |
| | 8 字节 | double Salary; |

图 6-6　结构体变量 s 和 t 的内存分配示意图

（4）访问结构体变量的下级成员。结构体变量中的每个成员都是一个变量，都具有各自的内存单元，可以单独访问它们。访问结构体变量下级成员的语法形式是“**结构体变量名.下级成员名**”，其中圆点“.”是**成员运算符**。例如：

```
cin >> s.ID >> s.Name >> s.Age >> s.Score;        // 输入结构体变量 s 各成员的信息
cout << s.ID << s.Name << s.Age << s.Score;       // 输出结构体变量 s 各成员的信息
```

细心的读者可能会发现，结构体类型与联合体类型在语法形式上非常相似，但这两者之间有着本质的区别。结构体变量的各成员都单独分配内存单元，分别保存各自的数据；而联合体变量各成员之间是共用内存的，同时只能保存一个成员的数据。它们的用途也不一样，结构体类型用于分类管理程序中的变量；联合体类型则是用于减少程序中变量对内存的占用。

结构体类型和数组类型都能保存多个数据，它们之间又有什么区别呢？一个数组变量只能保存多个同类型的数据，而一个结构体变量则可以保存多个不同类型的数据。

### 3. 结构体指针与结构体数组

#### 1）　结构体指针

可以定义结构体类型的指针变量来保存某个结构体变量的地址，然后通过指针变量间接访问该结构体变量及其成员。例如：

```
struct Student   a;                  // 定义一个 Student 类型的结构体变量 a
struct Student   *p;                 // 定义一个 Student 类型的结构体指针变量 p
p = &a;                              // 将 a 的地址赋值给同类型的指针变量 p
```

此时，指针变量 p 指向了结构体变量 a，则：

（1）*p 与 a 等价。

（2）(*p).ID 与 a.ID 等价，(*p).Name 与 a.Name 等价，(*p).Age 与 a.Age 等价，(*p).Score 与 a.Score 等价。

由于通过“(*p).”的形式访问结构体变量成员比较烦琐，C++语言引入了一种新的更加直观的运算符“->”，称为**指向运算符**。通过指向运算符访问结构体变量成员的语法形式是“**指针变量名->下级成员名**”。例如：

```
p->ID、p->Name、p->Age、p->Score
```

使用结构体指针变量需注意以下两点：

（1）指针变量与被访问的变量应具有相同的结构体类型。

（2）指针变量需先赋值，即先指向被访问的结构体变量，然后才能间接访问该变量。

**2）结构体数组**

可以定义结构体类型的数组变量，例如：

```
struct Student   x[10];                    // 定义一个 Student 类型的结构体数组 x，包含 10 个元素
```

结构体数组中的每个元素都是一个结构体变量，可以用下标访问数组元素及其下级成员。例如：

```
for (int n = 0; n <= 9; n++)                // 整体输入结构体数组 x 的信息
    cin >> x[n].ID >> x[n].Name >> x[n].Age >> x[n].Score;        // 输入数组元素 x[n]的信息
```

同样也可以用指针变量遍历结构体数组元素，例如：

```
struct Student   *p;                        // 定义一个 Student 类型的结构体指针变量 p
for (p = &x[0]; p <= &x[9]; p++)            // 通过指针变量 p 遍历数组 x 的元素
    cin >> p->ID >> p->Name >> p->Age >> p->Score;              // 输入当前数组元素的信息
```

### 4. 使用 typedef 命名结构体类型

可以用 typedef 为结构体类型命名，这样在定义变量时就不再需要 struct 关键字。例如：

```
typedef struct
{
    char ID[11], Name[9];                   // 保存学号和姓名的成员
    int Age;                                // 保存年龄的成员
    double Score;                           // 保存学分的成员
}   STUDENT;                                // 命名一个结构体类型 STUDENT
STUDENT   a, b;                             // 用 STUDENT 类型定义两个结构体变量 a, b
```

另外，定义结构体变量时可以初始化，例如：

```
STUDENT   a = { "12345", "张三", 18, 85 },   b = { "23456", "李四", 19, 90 };
```

## 本节习题

1. 有类型定义“typedef   int *   IPointer;”，则下列语句中正确的是（　　）。

   A. int x;   IPointer y = x;　　　　　　　B. int x;   IPointer *y = &x;

C. int x;　IPointer y = &x;　　　　D. double x;　IPointer y = &x;

2. 执行下列 C++语句：

```
typedef char   Name[5];
Name   x;
cout << sizeof( x );
```

执行结束后，显示器将显示（　　）。

A. 1　　　B. 4　　　C. 5　　　D. 语法错误

3. 定义枚举类型“enum ABC　{ one, two, three };”，则下列语句中正确的是（　　）。

A. enum ABC x = Two;　　　　B. enum ABC x = two;

C. enum ABC x = four;　　　　D. enum ABC x = 1;

4. 定义结构体类型“struct ST　{ char a; int b; };”，则下列语句中正确的是（　　）。

A. Struct St x;　　　　B. struct st x;

C. STRUCT ST X;　　　　D. struct ST X;

5. 定义结构体类型 ST 和变量 s“struct ST　{ int a; double b; }　s; ”，则下列语句中正确的是（　　）。

A. cin >> s;　　　　B. cin >> s.a >> s.b;

C. cin >> s{ a, b };　　　　D. cin >> s.c;

6. 定义一个结构体变量，该变量所占用的内存等于（　　）。

A. 各成员所需内存空间的总和

B. 各成员中占用内存最大者所需的内存空间

C. 结构体中第一个成员所占用的内存空间

D. 结构体中最后一个成员所占用的内存空间

## 6.6 结构化程序设计的应用与回顾

应用结构化程序设计方法，可以将一个大型程序分解成多个算法模块，分而治之。C++语言以函数的形式来描述各算法模块（即函数的定义），然后再通过调用关系将各模块组装起来（即函数的调用），最终形成一个完整的算法流程。一个编写好的 C++函数可以被同一项目中的多个程序员调用，也可以在今后的项目中继续使用，或者以公开销售的形式提供给任何其他程序员使用，这就是函数代码的重用。

同样，一个程序员可以调用其他程序员编写的函数，或调用以前项目中编写的函数，也可以调用 C++语言自身提供的系统函数。实际上，市场上还有很多厂家为程序员提供编写好的、可实现各种不同功能的**函数库**。调用这些函数库中的函数，程序员可以快速开发出各种功能强大的应用程序，这就是程序的**应用开发**。调用函数库中的函数就是重用这些函数的代码。代码重用极大地提高了软件开发的效率。

本节以微软公司开发的 Win32 API 函数库为例，具体介绍如何开发一个 Windows 图形用户界面（GUI）程序。读者可从中体会结构化程序设计方法是如何帮助程序员提高开发效率的。

结构化程序设计方法为团队分工协作和代码重用，开发大型软件系统提供了一种科学有效的方法。但后来为什么又出现了面向对象程序设计方法呢？本节将以回顾—总结—分析的方式为读者讲解结构化程序设计方法的不足。结构化程序设计的不足就是面向对象程序设计产生的背景。

## 6.6.1 开发 Windows 图形用户界面程序

如何开发一个 Windows 图形用户界面（GUI）程序？例如，如何画一个窗口或添加一个按钮呢？运用之前所学的 C++语言知识，你怎么也想不出如何开发这样的程序。是的，绝大部分程序员都不知道如何开发这样的程序。

Windows 是微软开发的。为了帮助程序员开发 Windows 程序，微软公司运用结构化程序设计方法，将画窗口或添加按钮这样的功能编写成一个个函数，总共有两千多个函数。将这些函数打包在一起形成一个函数库，这个函数库被称为 Win32 API。Win32 API 函数库随 Visual C++ 6.0 或 Visual Studio 系列集成开发环境提供。

程序员使用 Visual C++ 6.0 并调用 Win32 API 中的相关函数，就可以很容易地编写出一个具有图形用户界面的 Windows 程序，例如编写一个如图 6-7 所示的温度换算程序。假设用户在【摄氏温度】后面的编辑框中输入摄氏温度 10，单击【转换】按钮，程序将执行温度转换算法，并将转换得到的华氏温度 50.0 显示在【华氏温度】后面的文本框中。如果用户单击了【退出】按钮，则关闭程序窗口，退出程序。




图 6-7 具有图形用户界面的温度换算程序

Windows 图形用户界面程序的编程原理是：

（1）程序员首先使用图形界面设计器来设计窗口，设计好的窗口被称为**资源**。

（2）程序员编写**主函数**，按照上一步的设计结果创建并在桌面上显示窗口，然后等待用户操作。

（3）计算机可同时运行多个程序，桌面上将显示多个程序窗口。Windows 操作系统负责监控鼠标并捕获用户的操作，根据鼠标位置来判断用户对哪个程序窗口进行了操作。如果用户对某个窗口进行了操作（例如单击了某个按钮），那么 Windows 操作系统将自动调用该窗口所对应的处理函数（被称为**窗口过程**）。程序员应当为每个窗口编写一个窗口过程函数。

（4）Windows 操作系统在调用窗口过程函数时会以**消息**（message）的形式传递参数。

消息参数以整数编号来标记用户所做的不同操作。为方便程序员，Win32 API 将这些整数编号定义成易于记忆的符号常量。例如，符号常量 WM_COMMAND 表示用户单击了某个按钮或菜单，WM_CLOSE 表示用户单击了关闭窗口的叉号“×”。Windows 操作系统调用某个窗口过程并传递消息参数，这被称为是向窗口过程发送消息。窗口过程接收消息然后进行处理，这被称为是对消息的响应。

例如，用户执行图 6-7 所示的温度换算程序，计算机将弹出程序窗口，该窗口是由程序员在主函数中编写语句来创建的。假设用户输入摄氏温度后单击【转换】按钮，Windows 操作系统将立即捕捉该操作，并自动调用对应的窗口过程函数。调用时传递一个 WM_COMMAND 消息。**请注意**：窗口过程不是程序员在主函数中编写函数调用语句来调用的，而是由 Windows 操作系统自动调用的。像窗口过程这样由 Windows 操作系统调用的函数被称为**回调函数**（callback function）。窗口过程函数根据所接收到的消息参数可以判断出用户单击了【转换】按钮，按照程序功能应当执行温度转换算法并显示换算结果，因此程序员应当编写实现温度转换算法的程序代码。

使用 Visual C++ 6.0 编写上述温度换算程序，编程过程需分如下 3 步完成。

**第 1 步**：新建一个 Win32 Application 工程。**注**：之前我们编写命令行界面程序新建的是 Win32 Console Application 工程（即控制台应用程序）。

**第 2 步**：使用 Visual C++ 6.0 的图形界面设计器来设计窗口界面。图形界面设计器提供很多被称作控件的图形元素，例如静态文本框、编辑框和按钮等，程序员可以用拖拽的方法在窗口中添加这些图形元素。程序员为界面中的图形元素分别指定不同的整数编号，并用符号常量来表示（如图 6-8 所示）。例如，添加一个用于输入摄氏温度的编辑框，并用符号常量 IDC_EDIT_CTEMP 作为其整数编号。

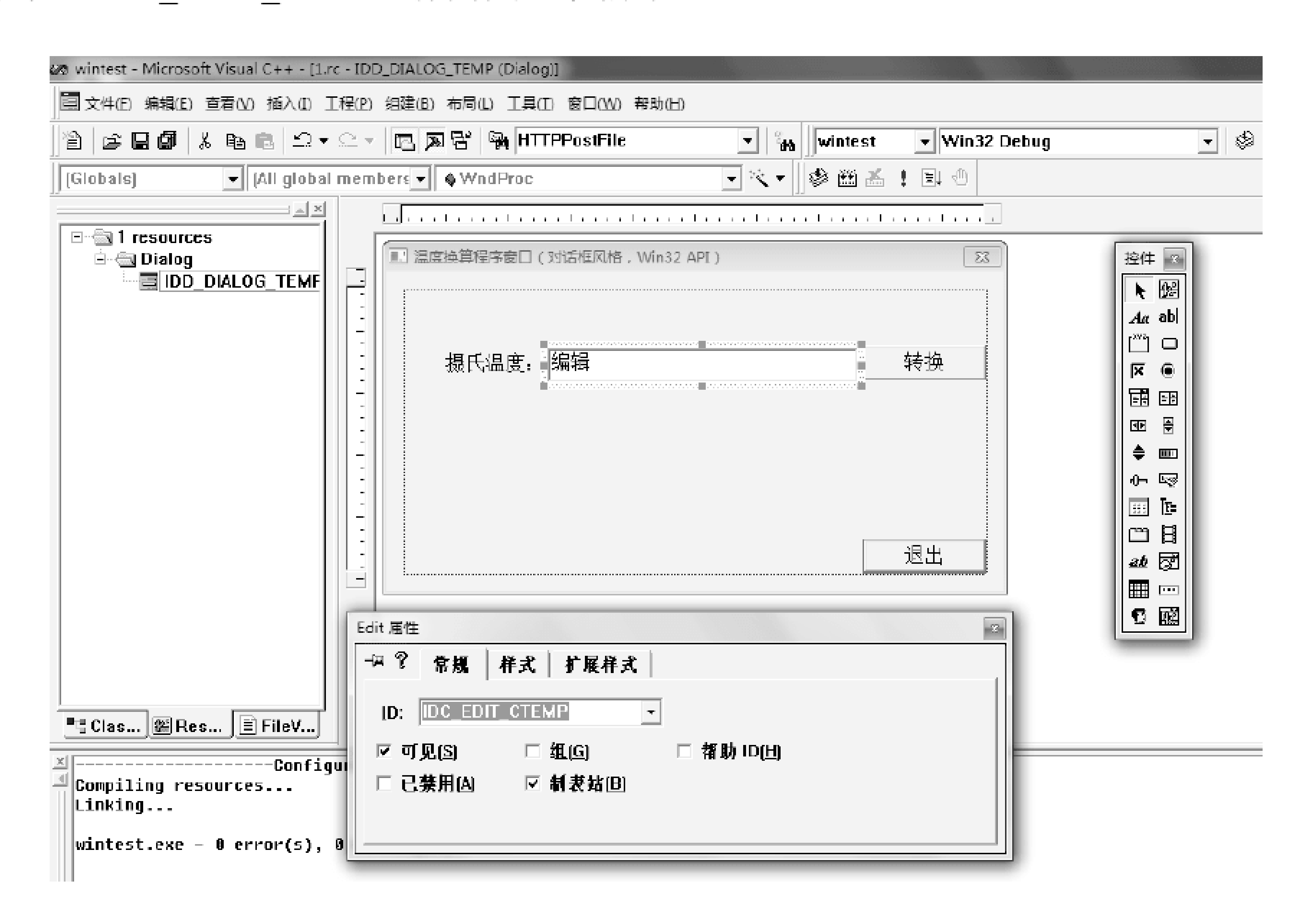



图 6-8　Visual C++ 6.0 的图形界面设计器

图形界面设计器将设计结果保存到一个资源文件（扩展名为.rc）中。Windows 操作系统将图形界面相关的元素统称为**资源**（resource）。图形界面设计器还将各界面元素所对应的符号常量定义在头文件 resource.h 中。图 6-8 设计结果所保存的资源文件和头文件如例 6-20 所示。

**例 6-20　图 6-8 设计结果所保存的资源文件和头文件**

**1. 资源文件**（.rc）的示例代码

```
#include "resource.h"
#include "afxres.h"
…  // 省略部分代码
/////////////////////////////////////////////////////////////////////////////
// 对话框窗口（用符号常量 IDD_DIALOG_TEMP 来标识）的设计结果
IDD_DIALOG_TEMP DIALOG DISCARDABLE   0, 0, 249, 129
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "温度换算程序窗口（对话框风格，Win32 API）"
FONT 10, "System"
BEGIN
    LTEXT              "摄氏温度：",IDC_STATIC,24,33,41,8
    EDITTEXT           IDC_EDIT_CTEMP,65,31,125,14,ES_AUTOHSCROLL
    LTEXT              "",IDC_STATIC_FTEMP,25,56,165,8
    PUSHBUTTON         "转换",IDC_BUTTON_CALC,192,30,50,14
    DEFPUSHBUTTON      "退出",IDC_BUTTON_QUIT,192,108,50,14
END
```

**2. 头文件**（resource.h）的示例代码

```
#define IDD_DIALOG_TEMP     101     // 温度换算程序窗口的编号
#define IDC_EDIT_CTEMP      1000    // 输入摄氏温度的编辑框的编号
#define IDC_STATIC_FTEMP    1001    // 显示华氏温度的文本框的编号
#define IDC_BUTTON_CALC     1002    // 【转换】按钮的编号
#define IDC_BUTTON_QUIT     1003    // 【退出】按钮的编号
```

**第 3 步：**新建一个 C++源程序文件，并编写温度换算程序的完整 C++代码（例 6-21）。Windows 图形用户界面的 C++程序结构主要分为两部分，一是主函数，二是窗口过程。

**例 6-21　温度换算程序的 C++代码（Win32 API）**

```
// 使用 Win32 API 函数库，编写一个将摄氏温度换算成华氏温度的图形用户界面程序
#include <windows.h>   // 插入声明 Win32 API 函数库的头文件 windows.h
#include "resource.h"  // 插入定义界面元素符号常量的头文件 resource.h
#include <stdio.h>  // 插入标准输入/输出函数的头文件 stdio.h

long CALLBACK WndProc(HWND hDlg, unsigned int message, unsigned int wParam, long lParam)
{
        switch (message)   // 处理用户不同的操作（即不同的消息）
        {
                case WM_COMMAND:    // 处理【命令】消息
```

```
11                  if (LOWORD(wParam) == IDC_BUTTON_CALC)
12                  {   // 处理【转换】按钮消息，转换温度
13                      HWND hEditCTemp = GetDlgItem(hDlg, IDC_EDIT_CTEMP);
14                      char str[50];
15                      GetWindowText(hEditCTemp, str, 50);
16                      double ctemp;
17                      sscanf(str, "%lf", &ctemp);
18                      HWND hStaticFTemp = GetDlgItem(hDlg, IDC_STATIC_FTEMP);
19                      double ftemp = ctemp*1.8 + 32;
20                      sprintf(str, "华氏温度：%6.1lf", ftemp);
21                      SetWindowText(hStaticFTemp, str);
22                      return 1;
23                  }
24                  else if (LOWORD(wParam) == IDC_BUTTON_QUIT)
25                  {   // 处理【退出】按钮消息，退出程序
26                      EndDialog(hDlg, LOWORD(wParam));    return 1;
27                  }
28                  break;
29              case WM_INITDIALOG:     // 处理【初始化】消息
30                  return 1;           // 可在此初始化对话框
31          }
32      return 0;
33  }
34
35  // 定义 Windows 图形用户界面程序的主函数 WinMain( )
36  int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrev, LPSTR lpLine, int nShow)
37  {
38          DialogBox(hInstance, (LPCTSTR)IDD_DIALOG_TEMP, NULL, (DLGPROC)WndProc);
39          return 1;
40  }
```

**主函数**的主要功能是创建窗口，并指定该窗口所对应的窗口过程。C++语言规定主函数的函数名必须是 main。VC 6.0 对此做出如下修改：Windows 图形用户界面程序的主函数名改为 WinMain。例 6-21 中，代码第 35~40 行为主函数 WinMain 的完整定义代码，其中最重要的语句是第 38 行的函数调用语句：

```
DialogBox(hInstance,  (LPCTSTR)IDD_DIALOG_TEMP,  NULL,  (DLGPROC)WndProc);
```

DialogBox 是 Win32 API 函数库提供的一个函数，其功能是按照资源文件中 IDD_DIALOG_TEMP 的设计来创建一个对话框风格的窗口，并指定其对应的窗口过程为 WndProc 函数。计算机执行 DialogBox 函数，将在桌面上创建一个图形窗口，这样程序用户就可以使用鼠标在窗口中进行操作了。至此，主函数 WinMain 的工作就完成了，下面的工作交由窗口过程函数 WndProc 继续完成。

**窗口过程函数**的功能是接收消息参数、判断消息种类，根据程序任务要求为不同消息设计不同的处理算法。例 6-21 中代码第 6~33 行就是窗口过程函数 WndProc 的完整定义代码。

假设用户在【摄氏温度】编辑框中输入 10，然后单击【转换】按钮（其对应的编号为

1002，即符号常量 IDC_BUTTON_CALC），则 Windows 将自动调用函数 WndProc。调用时，形参 message 将接收到消息 WM_COMMAND，形参 wParam 接收到【转换】按钮的编号（即符号常量 IDC_BUTTON_CALC）。程序员使用 switch-case 或 if-else 语句来判断消息。WM_COMMAND 和 IDC_BUTTON_CALC 这两个参数表明用户单击了【转换】按钮。程序员根据程序功能要求，在用户单击【转换】按钮时编写算法代码（例 6-21 中代码第 12~23 行）：首先取出摄氏温度编辑框（即 IDC_EDIT_CTEMP）中的数据（字符串类型），转成 double 型数值，然后计算其对应的华氏温度，再转成字符串形式并显示在华氏温度文本框（即 IDC_STATIC_FTEMP）中。这样，用户输入摄氏温度 10，单击【转换】按钮，就可以在华氏温度文本框中看到换算结果 50.0 了。

假设用户单击【退出】按钮（即 IDC_BUTTON_QUIT），则执行下列语句：

```
EndDialog(hDlg,  LOWORD(wParam));
```

EndDialog 也是 Win32 API 函数库提供的一个函数，其功能是关闭窗口并退出程序。

上面通过温度换算程序的例子简单介绍了 Windows 图形用户界面程序的开发过程，其中的某些语法细节没有展开，但这不影响对程序的理解。可以看出，只要使用 Win32 API 函数库，程序员就可以快速开发出 Windows 图形用户界面程序。以函数形式组织和重用代码可以极大地提高程序开发效率，这就是结构化程序设计思想的精髓所在。

在学习完结构化程序设计之后，程序员在拿到一个具体的程序设计任务时，首先应当考虑有哪些现成的函数库可以用。使用现成的函数库开发程序，开发周期将大大缩短。如果实在找不到所需要的函数库，再考虑自己从零开始编写程序代码。基于已有的函数库开发程序，相当于是用别人已经做好的零件来组装产品。程序的应用开发，通常就是用已有的程序零件来组装自己的软件产品。

## 6.6.2 结构化程序设计回顾

本节首先回顾一下 C++语言中的结构化程序设计方法。

### 1. 结构化程序设计回顾

（1）**简单程序**。C++语言通过常量或变量存储数据，每个数据都有特定的数据类型。通过 cin、cout 指令来输入/输出数据。通过运算符编写表达式对数据进行计算和处理。通过控制语句描述不同的算法结构，实现比较复杂的算法。

（2）**复杂程序**。结构化程序设计方法通过划分模块来实现更长、更复杂的处理算法。C++语言支持结构化程序设计方法，以函数的语法形式来描述和组装算法模块，即函数的定义和调用。一个复杂的 C++程序可能包含很多函数，源代码会很长。将函数分散保存在不同的源程序文件中，这就是一个多文件结构的 C++程序。

（3）**复杂数据**。程序设计任务可能会面临比较复杂的数据。C++语言通过自定义数据类型来描述复杂数据。常用的自定义数据类型有数组、枚举类型、联合体和结构体等。

### 2. 结构化程序设计的基本思想

“程序=数据+算法”，变量是程序中的数据元素，函数是程序中的算法元素。结构化程序设计方法的基本指导思想可归纳成如下 3 点。

（1）**模块化设计**。将算法划分成模块，分而治之。用函数描述算法模块。在确定好函数功能及调用接口之后，程序员各自编写自己的函数，互不干扰，并行开发。模块化设计是团队分工协作开发的基础，开发大型程序必须采用团队分工协作的开发模式。

（2）**重用函数代码**。函数是一段相对独立的代码，能完成某种特定的功能。通常将相对独立、经常使用的功能提炼出来，编写成函数。函数可以被多次调用，或被多个程序员使用，或在多个项目中使用，这就是函数代码的重用。代码重用实现了**“一次开发，长期使用”**，这极大地提高了程序开发效率。函数是结构化程序设计实现代码重用的基本形式。

（3）**分类管理数据**。结构体类型对数据进行分类管理。它将具有内在关联关系的变量组合在一起形成一个逻辑上的整体，使变量成为整体的下属成员。使用结构体可以将程序中大量的数据元素按其内在关联关系划分成一个个相对独立的整体再进行管理，这体现了一种朴素的**“分类管理”**思想。分类可以更好地管理程序代码。

### 3. 结构化程序设计所面临的问题

结构化程序设计方法为团队分工协作，开发大型软件提供了一种科学有效的方法。历史上许多大型的软件系统，例如操作系统、数据库管理系统等，都是采用结构化程序设计方法开发出来的。

在对这些大型软件系统的后续维护、升级过程中，人们发现对程序的修改非常困难，甚至无法完成。其主要原因是因为结构化程序设计方法中的数据与算法是分离的，同时它们之间的耦合性还非常强。结构化程序设计中的算法被分解成多个算法模块，即多个函数。如果只修改其中的某个算法模块，程序员找到其对应的函数代码，修改并重新编译连接就可以了。但如果想要修改数据，例如修改数据的类型、添加或减少数据项等，程序员除了修改对应的变量定义语句之外，还需要修改所有与之关联的函数。大型软件系统包含成千上万个函数，这些函数存在多层嵌套调用关系，每层调用之间都可能通过形实结合来传递数据。程序员需要修改所有与数据关联的函数定义、声明或调用代码，这是一项非常艰巨的任务。

解决上述问题的方法就是在分解程序模块时，不是单纯分解算法模块，而是将数据和与之关联的处理算法划分在一起形成**“数据类”**。数据类是数据及其处理算法的完整描述，数据类等于“数据+算法”。数据类中变量和函数的定义代码被集中在一起。无论是数据或算法，修改时通常只要修改数据类的定义代码即可，与其他代码无关。面向对象程序设计方法就是按照数据类来分解和编写程序的。

### 4. 面向对象程序设计方法

**面向对象程序设计**正是基于“分类管理”的思想，在结构化程序设计基础之上所做的进一步发展。面向对象程序设计方法将程序中的数据元素和算法元素按其内在关联关系统一进行分类管理，这就形成了**“类”**。类相当于是一种自定义数据类型，用类所定义的变量

称为“**对象**”。支持面向对象程序设计的计算机语言使用类的语法形式来定义数据类，然后通过定义对象来使用数据类。在团队分工协作开发模式中，某些程序员定义类，编写类代码；某些程序员使用类定义对象，然后通过对象重用别人的类代码。

面向对象程序设计可以更好地组织和管理程序代码，更便于代码重用，更适用于团队分工协作，能更容易地开发出质量更好、功能更强大的程序。下一章开始，我们将正式走入面向对象的世界！

### 学习本章的要点

■ 读者要掌握与多文件结构相关的语法知识，其中包括外部函数和全局变量的声明、头文件等。

■ 读者要重点掌握带默认形参值的函数、重载函数和内联函数这三种常用函数形式。

■ 读者要牢固树立重用代码的思想，学会通过调用别人编写的函数来提高开发效率。

## 6.7 本章习题

1. **阅读程序**。阅读下列 C++程序（共两个文件）。阅读后请说明各程序文件及函数的功能，并对每条语句进行注释，说明其作用。

```
// 程序文件：1.cpp
#include <iostream>
using namespace std;
void fun(int x)
{
	if (x < 0)
	{
		cout << '–';    x = –x;
	}
	while (x != 0)
	{
		cout << x%10;    x /= 10;
	}
	cout << endl;
}
void fun(char *str)
{
	int N = 0;
	while (str[N] != '\0')    N++;
	for (int n = N–1; n >= 0; n– –)    cout << str[n];
	cout << endl;
}
// 程序文件：2.cpp
extern void fun(int x);
extern void fun(char *);
int main( )
```

```
{
    fun(–2015 );
    fun( "–2015" );
    return 0;
}
```

2. **阅读程序**。阅读下列 C++程序。阅读后请说明程序的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
using namespace std;
#define FUN(T, X, Y)   {   T = X;   X = Y;   Y = T;   }
int main( )
{
    int x, y, z;
    cin >> x >> y;
    FUN( z, x, y );
    cout << x << ", " << y << endl;
    return 0;
}
```

3. **程序改错**。阅读下列 C++程序，并检查其中的语法错误。修改错误，并保证程序的功能不变。

```
#include <iostream>
using namespace std;
void fun(char *str, char ch = '#')              // 删除字符串 str 中所有的字符 ch
{
    int m = 0, n;
    while (str[m] != '\0');                     // 循环 1：依次检查字符串 str 中字符
    {
        if (str[m++] != ch)                     // 如果当前字符不等于字符 ch，则继续下一次循环
            break;                              // 中止本次循环，进入下一次循环
        n = m;
        do                                      // 循环 2：删除当前字符，就是依次将其后续字符前移一位
        {
            str[n-1] = str[n];   n++;           // 将后续字符依次前移一位
        } while (str[n-1] != '\0')              // 遇到字符串结束符时结束循环 2
    }
}
int main( )
{
    char name[ ] = "My*#Name*#Is*#John";
    fun( name );                                // 调用函数 fun 删除 name 中的井号#
    cout << name << endl;             // 显示删除井号后的 name，应显示：My*Name*Is*John
    fun( name );                                // 再次调用函数 fun 删除 name 中的星号*
    cout << name << endl;                       // 显示删除星号后的 name，应显示：MyNameIsJohn
    return 0;
}
```

4. **程序改错**。阅读下列 C++程序，并检查其中的语法错误。修改错误，并保证程序的功能不变。

```
#include <iostream>
using namespace std;
double Power( double x, int n )              // 定义递归函数求 x 的 n 次方，要求 n 为非负整数
{
    double result;
    if (n = 0)   result = 1;                 // 递归终结条件：n = = 0，此时 x 的 0 次方等于 1
    else   result = x * Power(x, n);         // 递归公式
    return result;
}
int main( )
{
    double x, value;   int n;
    cin >> x >> n;
    if (n >= 0)                              // 如果指数 n 为非负整数
        value = Power( x, n );               // 则调用函数 fun 求 x 的 n 次方
    else                                     // 如果指数 n 为负整数
        value = 1.0 / Power( x, n );         // 即求 x 的–n 次方的倒数
    cout << value << endl;                   // 显示结果
    return 0;
}
```

5. **编写程序**。编写一个求圆面积的 C++程序。要求：使用内联函数的形式编写求圆面积的子函数 Area，主函数 main 负责输入半径、调用子函数 Area 求面积，并显示面积的计算结果。

6. **编写程序**。模仿例 6-14，编写一个绘制余弦函数 cos($x$)波形的 C++程序。

# 第7章 面向对象程序设计之一

“程序=数据+算法”，变量是程序中的数据元素，函数是程序中的算法元素。**面向对象程序设计方法**将程序中的数据元素和算法元素根据其内在的关联关系进行分类管理，这就形成了**类**的概念。类是一种由程序员定义的高级数据类型，用类所定义的变量称为**对象**。

基于类与对象编程可以更好地组织和管理程序代码，也更便于代码重用。在团队分工协作开发模式中，某些程序员定义类，编写类代码；某些程序员使用类定义对象，然后通过对象重用类代码。

## 7.1 面向对象程序设计方法

程序是用于处理数据的，通常应包括如下 4 项功能。

（1）定义保存数据的变量。

（2）输入原始数据。

（3）处理数据。

（4）输出处理结果。

其中，（2）和（4）所完成的输入/输出功能是程序提供给用户的交互界面，简称为**用户界面**。

本节通过一个程序实例，直观介绍结构化程序设计是如何演变到面向对象程序设计的，然后在程序演变的过程中具体讲解什么是面向对象的程序设计方法。

**程序实例：**编写一个计算长方形面积和周长的 C++演示程序。程序由甲、乙两位程序员分工协作，共同编写。

### 7.1.1 结构化程序设计中的函数

可以使用结构化程序设计方法，将程序划分成 3 个函数。两位程序员分工协作，甲负责编写主函数，乙负责编写计算面积和周长的子函数。例 7-1 给出完整的计算长方形面积和周长的 C++程序代码。

**例 7-1　计算长方形面积和周长的 C++程序代码（函数）**

**程序员甲：主函数（1.cpp）**

```
#include <iostream>
using namespace std;

#include "2.h"   // 插入头文件 2.h

int main( )
{
      int a, b;   // 定义保存长宽数据的变量
      cin >> a >> b;   // 输入长方形的长和宽

      cout << Area(a, b) << endl;   // 长方形面积
      cout << Len(a, b) << endl;   // 长方形周长
      return 0;
}
```

**程序员乙：子函数（2.cpp）**

```
// 计算长方形面积和周长的函数
int Area(int length, int width)
{   return ( length*width );   }
int Len(int length, int width)
{   return ( 2*(length+width) );   }
```

**程序员乙：头文件（2.h）**

```
// 声明外部函数的原型
int Area(int length, int width);
int Len(int length, int width);
```

例 7-1 程序的代码说明如下。

**1）程序员甲负责编写主函数 main**

主函数中，程序员甲定义了两个保存长方形长宽数据的变量 a、b，通过 cin 指令接收用户输入的长宽值，然后调用子函数 Area 和 Len 分别计算出长方形的面积和周长，并通过 cout 指令将计算结果反馈给用户。其中的 cin 和 cout 指令为用户提供了一种命令行风格的交互界面。

程序员甲编写的主函数代码共完成了 4 项程序功能中的 3 项，即定义保存数据的变量、输入原始数据和输出处理结果。剩余的一项功能（即处理数据）是由程序员乙编写子函数代码来完成的。程序员甲在计算长方形面积和周长时通过调用子函数就轻松完成了数据处理功能。换句话说，程序员乙帮程序员甲分担了部分编程工作。

**2）程序员乙负责编写子函数**

程序员乙负责编写两个子函数，子函数 Area 的功能是计算长方形面积，子函数 Len 的功能是计算长方形周长。程序员乙编写子函数时定义了两个形参 length 和 width，用于接收主函数传递过来的长宽值（即存放在实参 a、b 中的值），然后编写算法代码求出长方形的面积或周长，并以返回值的形式将结果返回给主函数。程序员乙将计算长方形面积或周长的算法代码定义成函数，程序员甲调用该函数就能轻松实现相应的程序功能。

调用函数实际上是重用该函数的算法代码，实现其规定的程序功能。程序员乙编写好的函数 Area 和 Len 可以在本次项目中使用，也可以在今后的项目中使用，或提供给任何其他程序员使用。无论是什么项目，或是哪位程序员，只要调用这两个函数就都能轻松实现求长方形面积或周长的功能。将算法代码定义成函数的好处是“一次编写，长期使用”。

是的，也许有人会觉得重用这两个求长方形面积或周长的算法代码没有什么价值，自己也能编写。但设想一下，如果这是个 JPEG 图像压缩算法呢？本例中，求长方形面积或

周长仅仅是个例子，函数才是读者应该关注的重点。

结构化程序设计中，函数是重用**算法代码**的基本语法形式。

**3）两类不同的程序员角色**

代码重用可以减少开发工作量，提高软件质量。代码重用过程中，程序员有两类角色，一类是提供代码的程序员（**代码提供者**），另一类是使用代码的程序员（**代码使用者**）。

例 7-1 程序中，计算长方形面积和周长的子函数 Area、Len 是被重用的代码。程序员乙编写重用代码，是代码提供者。程序员甲编写主函数时调用子函数 Area 和 Len，是代码使用者。**注：**名词“**重用代码**”是指被重用的代码。

实际应用中，重用代码经常是由一位程序员编写，然后被多个程序员使用。也就是说，类似例 7-1 程序中程序员乙的角色只有一个人，而程序员甲的角色则有很多人（见图 7-1）。

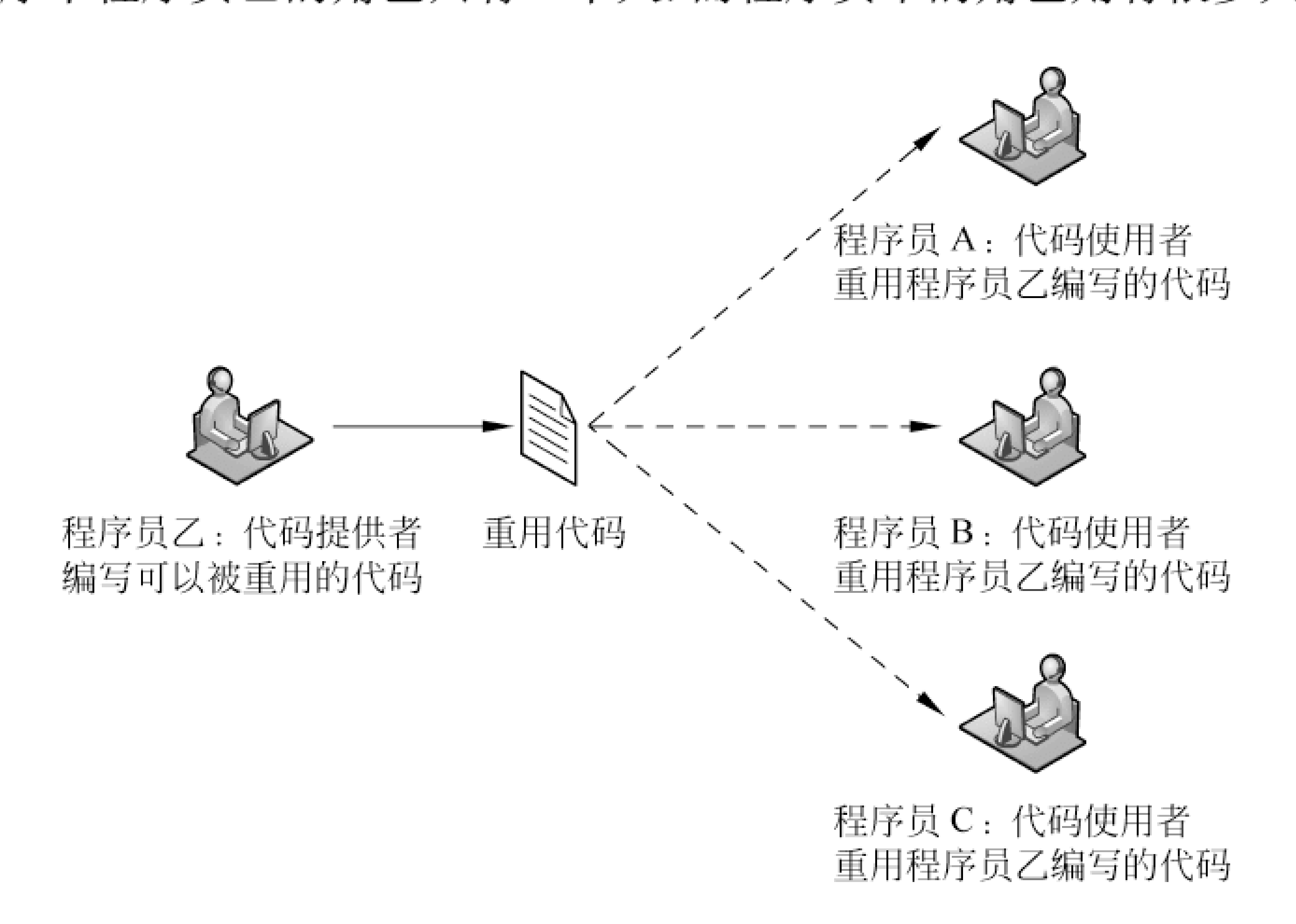



图 7-1　代码重用过程中的两类程序员角色

如何让重用代码完成更多的程序功能，这是软件技术不懈追求的目标。如果程序员乙编写的重用代码能多完成一项功能，那么众多使用代码的程序员就都能少承担一项功能，这就从整体上提高了软件开发的效率。**请注意**，增加重用代码的功能可以减少**代码使用者**的工作量，但反过来会增加**代码提供者**的工作量。编写重用代码的程序员应当具备“辛苦我一个，幸福千万家”的精神。

在例 7-1 中，程序员甲承担了 4 项程序功能中的 3 项（即定义保存数据的变量、输入原始数据和输出处理结果），而程序员乙只承担了一项数据处理功能。下面我们将基于这个例子来演示，如何将程序员甲当前所承担的 3 项程序功能一步一步地继续转移给程序员乙。

## 7.1.2　结构化程序设计中的结构体类型

结构体类型将多个具有内在关联关系的变量组合在一起形成一个逻辑上的整体，变量成为整体的下属成员。定义好的结构体类型将被当作一种新的数据类型来定义变量，所定义出的变量称为结构体变量。

程序员乙通过定义长方形结构体类型可以帮程序员甲再分担一项程序功能，即定义保

存数据的变量。修改例 7-1，在程序中引入结构体类型。例 7-2 给出修改后的计算长方形面积和周长的 C++程序代码。

**例 7-2 计算长方形面积和周长的 C++程序代码（结构体类型）**

**程序员甲**：主函数（1.cpp）

```
#include <iostream>
using namespace std;

#include "2.h"   // 插入头文件 2.h

int main( )
{
    // int a, b;   // 删除该定义变量语句
    // 改用结构体类型 Rectangle 定义变量
    struct Rectangle   rect;   // 定义结构体变量
    cin >> rect.a >> rect.b;
    // 用 rect 的下级成员 a 保存长度
    // 用 rect 的下级成员 b 保存宽度

    // 调用子函数求长方形的面积和周长
    cout << Area( rect.a, rect.b ) << endl;
    cout << Len( rect.a, rect.b ) << endl;
    return 0;
}
```

**程序员乙**：子函数（2.cpp）

```
// 计算长方形面积和周长的函数
int Area(int length, int width)
{   return ( length*width );   }
int Len(int length, int width)
{   return ( 2*(length+width) );   }
```

**程序员乙**：头文件（2.h）

```
// 定义一个长方形结构体类型
struct Rectangle
{
    int a;   // 保存长度的成员 a
    int b;   // 保存宽度的成员 b
};

// 声明外部函数的原型
int Area(int length, int width);
int Len(int length, int width);
```

例 7-2 程序的代码说明如下。

**1）程序员乙定义结构体类型 Rectangle**

变量 a、b 分别保存长方形的长度和宽度。它们都属于长方形数据的一部分，本来就是一个逻辑上的整体。程序员乙将这两个具有内在关联关系的变量组合在一起，在头文件 2.h 中定义一个长方形结构体类型 Rectangle，变量 a、b 成为其下属成员。

和基本数据类型相比，结构体类型是一种由程序员定义出的复杂数据类型，其中可以包含多个变量成员。定义结构体类型就是声明其中包含了哪些变量成员，以及这些变量成员的数据类型。

**2）程序员甲使用结构体类型 Rectangle 定义变量**

定义好的结构体类型将被当作一种新的数据类型来定义变量，所定义出的变量称为结构体变量。程序员甲使用结构体类型 Rectangle，所定义出的变量 rect 就是一个结构体变量。

结构体变量 rect 是一个复杂变量，其中包含两个下级成员，即长度 a 和宽度 b。程序员甲使用这两个下级成员 rect.a 和 rect.b 来分别保存长方形的长度和宽度。

**3）理解结构体类型**

定义结构体类型的代码描述了一种更高层次上的数据类型。和 int、double 等**基本数据类型**相比，结构体类型是一种由程序员定义的**高级数据类型**。

在结构化程序设计中，函数所描述的是一种算法代码。调用函数就是重用算法代码，实现其规定的算法功能。结构体类型的定义代码中包含的是一组定义变量语句，这是一种

数据代码。使用结构体类型定义变量，就是重用数据代码，实现其规定的数据管理功能。结构体类型是结构化程序设计中重用**数据代码**的基本语法形式。

结构体类型将程序中大量的数据元素按其内在关联关系划分成一个个相对独立的整体再进行管理，这体现了一种朴素的“**分类管理**”思想。分类可以更好地管理程序代码。

### 7.1.3　面向对象程序设计中的分类

面向对象程序设计正是基于“分类管理”的思想，在结构化程序设计基础之上的进一步发展。面向对象程序设计方法将程序中的数据元素和算法元素按其内在关联关系统一进行分类管理，这就形成了“**类**”。类是结构体类型的进一步扩展，它既可以包含变量，又可以包含函数。类是整体，变量和函数是类的下属成员，分别称为**数据成员**和**函数成员**。同结构体类型一样，定义好的类将被当作一种新的数据类型来定义变量，用类所定义的变量被称为**对象**。

修改例 7-2，在程序中引入类的概念。程序员乙在长方形结构体类型的基础上，进一步将与长方形相关的 2 个函数（Area、Len）也包含进来，使用关键字 class 定义一个长方形类 Rectangle。例 7-3 给出修改后的计算长方形面积和周长的 C++程序代码。

**例 7-3　计算长方形面积和周长的 C++程序代码（类与对象）**

**程序员甲**：主函数（1.cpp）

```
#include <iostream>
using namespace std;

#include "2.h"   // 插入头文件 2.h

int main( )
{
    // ~~struct Rectangle   rect;~~  // 删除该语句
    // 改用类 Rectangle 定义变量（即对象）
    Rectangle   rect;  // 定义一个长方形对象 rect
    cin >> rect.a >> rect.b;
    // 用 rect 的数据成员 a 保存长度
    // 用 rect 的数据成员 b 保存宽度

    // 调用 rect 的函数成员求其面积和周长
    cout << rect.Area( ) << endl;
    cout << rect.Len( ) << endl;
    return 0;
}
```

**程序员乙**：类实现程序文件（2.cpp）

```
#include "2.h"   // 插入头文件 2.h
// 定义长方形类 Rectangle：类实现部分
// 给出各函数成员的完整定义代码
int Rectangle::Area( )
{   return ( a*b );   }
int Rectangle::Len( )
{   return ( 2*(a+b) );   }
```

**程序员乙**：类声明头文件（2.h）

```
// 定义一个长方形类 Rectangle
// 定义类的代码分为声明和实现两部分
class Rectangle   // 类声明部分
{
public:
    int a;   // 数据成员：保存长度
    int b;   // 数据成员：保存宽度
    int Area( );   // 函数成员：计算面积
    int Len( );   // 函数成员：计算周长
};
// 类 Rectangle 的实现部分放在.cpp 文件中
```

例 7-3 程序的代码说明如下。

**1）程序员乙定义长方形类 Rectangle**

程序员乙将与长方形相关的 2 个变量（a、b）和 2 个函数（Area、Len）组合在一起，定义一个长方形类 Rectangle。长方形类 Rectangle 是一个整体，变量 a、b 是其数据成员，

函数 Area 和 Len 是其函数成员。定义类的代码分为两部分，分别是**类声明**部分和**类实现**部分。程序员乙将 Rectangle 类声明部分的代码保存在头文件 2.h 中，将类实现部分的代码保存在程序文件 2.cpp 中。

- **类声明**部分（class declaration）。使用关键字 class 并指定类名，并在随后的大括号中声明所包含的数据成员和函数成员。声明函数成员就是声明其原型，完整的函数定义代码被放在类实现部分。**注：**类声明部分有一个关键字 public，其语法作用将在 7.1.4 小节中再做讲解。
- **类实现**部分（class implementation）。在类实现部分给出各函数成员的完整定义代码。定义时需在函数名前加类名“Rectangle::”进行限定，指明该函数是属于 Rectangle 类的。其中的“::”为两个冒号，被称为**作用域运算符，**或**作用域分辨符**。

在类定义中，数据成员（例如变量 a、b）相当于是类中的全局变量，函数成员可以直接访问它们。例如，函数 Area 和 Len 在计算长方形面积和周长时直接从数据成员 a、b 中读取长宽值，因此这两个函数不需要再定义形参来接收长宽数据。以类的形式来组织程序代码，可以有效减少函数间的参数传递。

从类的定义代码可以看出，类是一种由程序员定义的高级数据类型（被称为**类类型**），其中描述了类包含哪些数据成员以及各成员的数据类型（这属于数据代码），同时还以函数成员的语法形式描述了类具有哪些处理算法（这属于算法代码）。

**2）程序员甲使用长方形类 Rectangle 定义对象**

类是结构体类型与函数的结合体，其中既包含数据代码，又包含算法代码。类是一种由程序员定义的高级数据类型，用类所定义的变量称为**对象**。

程序员甲编写主函数时使用长方形类 Rectangle 定义一个对象 rect。rect 被称为是一个长方形类的对象。按照长方形类 Rectangle 的定义，对象 rect 将包含 2 个数据成员（即 rect.a 和 rect.b），程序员甲使用这两个数据成员来分别保存长方形的长度和宽度。对象 rect 还包含 2 个函数成员，即 rect.Area( )和 rect.Len( )，程序员甲调用这 2 个函数成员来分别计算长方形的面积和周长。

使用类定义对象，然后访问对象的成员，这实际上是重用该类的代码，实现其规定的程序功能。程序员乙编写好的长方形类 Rectangle 可以在本次项目中使用，也可以在今后的项目中使用，或提供给任何其他程序员使用。无论是什么项目，或是哪位程序员，只要使用这个类就都能轻松实现求长方形面积或周长的功能。类代码也是“一次编写，长期使用”。重用类代码时，既重用了数据代码，又重用了算法代码。在面向对象程序设计中，类是重用“**数据代码+算法代码**”的基本语法形式。

针对计算长方形面积和周长的问题，例 7-2、例 7-3 分别采用结构化程序设计方法和面向对象程序设计方法，进而设计出了不同的程序。这两种程序设计方法有什么不同之处呢？

程序设计方法主要应用于大型软件的设计开发。大型程序的功能很强，这意味着要处理大量的数据，数据处理的算法也很多、很复杂。**程序设计方法**就是研究如何对大型程序设计任务进行分解的方法，其**基本思想**是：将大型程序中的数据元素和算法元素分解成程序零件，将不同零件的设计任务交由不同的程序员完成，这样就能以团队的形式来共同开发，然后将开发好的零件组装在一起，最终完成复杂的程序功能。目前，程序设计方法分为结构化程序设计和面向对象程序设计两种，它们分别采用不同的方式来分解和组装程序零件。

### 1. 结构化程序设计方法

结构化程序设计方法将程序中的复杂算法分解成多个函数，函数是分解出的算法零件。结构化程序设计方法将大量保存数据的变量按其内在关联关系划分成一个个相对独立的结构体类型，结构体类型是分解出的数据零件。可以看出，结构化程序设计方法所分解出的算法零件和数据零件是分离的。这种分离的零件会带来什么后果呢？例 7-2 就是采用结构化程序设计方法分解出了相互分离的算法零件和数据零件，我们结合这个例子来做进一步分析。

例 7-2 将计算长方形面积和周长的算法分解成 2 个算法零件（即函数 Area 和 Len），将保存长方形长宽数据的变量 a、b 组合在一起形成 1 个数据零件（即结构体类型 Rectangle）。按这种方式所分解出的算法零件和数据零件在语法上是相互独立的（即相互分离的）。假设例 7-2 在编写完成后，程序员乙发现长方形结构体类型 Rectangle 中保存长宽数据的变量 a、b 被定义成了 int 型，只能处理整数。而实际应用中长方形的长宽数据经常是实数，程序员乙希望对这个数据零件进行升级，将变量 a、b 的数据类型修改为 double 类型。

程序员乙按如下形式修改头文件 2.h 中结构体类型 Rectangle 的定义代码：

```
struct Rectangle                    // 修改成员 a、b 的数据类型
{
    double a;                       // 原来为：int a;
    double b;                       // 原来为：int b;
};
```

将数据零件中的数据类型改为 double 后，程序员乙发现还需要继续修改算法零件，即修改函数 Area 和 Len，必须将这两个函数的形参和返回值类型同步改为 double 型。修改后的函数代码如下：

```
double Area(double length, double width)        // 原来为：int Area(int length, int width)
{   return ( length*width );   }
double Len(double length, double width)         // 原来为：int Len(int length, int width)
{   return ( 2*(length+width) );   }
```

可以看出，虽然数据零件和算法零件是相互分离的，但仍然互相耦合，互相影响。

因为函数 Area 和 Len 的定义代码改变了，其头文件 2.h 中对应的原型声明代码也要做相应修改。修改后的原型声明代码如下：

```
double Area(double length, double width);       // 原来为：int Area(int length, int width);
double Len(double length, double width);        // 原来为：int Len(int length, int width);
```

可以看出，如果想要修改结构化程序中的数据（例如修改数据的类型、添加或减少数据项等），程序员除了修改对应的变量定义语句之外，还需要修改所有与之关联的函数定义、声明或调用代码。大型软件系统包含成千上万个函数，其中还存在多层嵌套调用关系。这些函数是由不同程序员编写的，被分散保存在不同的程序文件中。修改数据所造成的连带修改范围会很广，也可能会涉及很多位程序员，修改难度将非常大。造成上述问题的原因就在于结构化程序设计方法将本来具有关联关系的数据和算法割裂开了。

2. 面向对象程序设计方法

为了解决结构化程序设计方法的不足，面向对象程序设计方法在分解程序零件时，不是单纯分解算法或数据，而是将数据和与之关联的算法划分在一起形成**“数据类”**。数据类是数据及其处理算法的完整描述，数据类等于“数据+算法”。数据类将具有关联关系的变量和函数集中定义在一起，无论是数据或算法，修改时通常只要修改该数据类的代码即可，与其他代码无关。面向对象程序设计方法就是按照数据类来分解和编写程序的。例如，例 7-3 中的长方形类 Rectangle 就是按上述方法所分解出的一个数据类。

在例 7-3 中，如果程序员乙希望对程序进行升级，将长方形类 Rectangle 中变量 a、b 的数据类型修改为 double 类型，则只要修改长方形类 Rectangle 中相关的代码。修改后长方形类 Rectangle 的定义代码如下：

```
class Rectangle                         // 修改头文件 2.h 中的类声明部分
{
public:
        double a;                       // 原来为：int a;
        double b;                       // 原来为：int b;
        double Area( );                 // 原来为：int Area( );
        double Len( );                  // 原来为：int Len( );
};
// 修改程序文件 2.cpp 中的类实现部分
double Rectangle::Area( )               // 原来为：int Rectangle::Area( )
{   return ( a*b );   }
double Rectangle::Len( )                // 原来为：int Rectangle::Len( )
{   return ( 2*(a+b) );   }
```

例 7-3 以类的形式来组织程序代码，其中的函数成员 Area 和 Len 都没有形参，修改时只需修改返回值类型即可，因此代码修改量减少了。更为重要的是，在修改类中数据成员时，其连带修改的代码一般不会超出本类的范围。一个类通常是由一位或少数几位程序员编写的，修改所牵涉的程序员也比较少。因此以类的形式来组织程序代码，今后的修改难度将大大降低。

### 7.1.4 面向对象程序设计中的封装

从例 7-1 演变到例 7-3，程序员乙已经承担了 4 项程序功能中的 2 项，即定义保存数据的变量和处理数据。程序员乙是编写重用代码的程序员，我们希望他的重用代码能完成更多的程序功能。下面我们继续挖掘程序员乙的潜能，让他帮助程序员甲完成剩余的 2 项程序功能，即输入原始数据和输出处理结果。

程序员乙继续在例 7-3 的长方形类 Rectangle 中添加函数成员 Input 和 Output，实现输入原始数据和输出处理结果的功能。例 7-4 给出添加函数成员后的计算长方形面积和周长的 C++程序代码。

**例 7-4 计算长方形面积和周长的 C++程序代码（添加输入/输出功能）**

**程序员甲：主函数（1.cpp）**

```
// #include <iostream>      // 删除这 2 条语句
// using namespace std;
// 主函数不再使用 cin/cout，删除上面 2 条语句
#include "2.h"    // 插入头文件 2.h

int main( )
{
    // 使用功能完善后的类 Rectangle 定义对象
    Rectangle   rect;   // 定义一个长方形对象 rect
    // cin >> rect.a >> rect.b;   // 删除该语句
    rect.Input( );   // 调用 Input 成员输入长宽
    // cout << rect.Area( ) << endl;   // 删除该语句
    // cout << rect.Len( ) << endl;   // 删除该语句
    rect.Output( );   // 调用 Output 成员输出结果
    return 0;
}
```

**程序员乙：类实现程序文件（2.cpp）**

```
#include <iostream>
using namespace std;
// 本程序需使用 cin/cout，添加上面 2 条语句
#include "2.h"    // 插入头文件 2.h
// 定义长方形类 Rectangle：类实现部分
double Rectangle::Area( )
{   return ( a*b );   }
double Rectangle::Len( )
{   return ( 2*(a+b) );   }
void Rectangle::Input( )   // 输入长宽
{   cin >> a >> b;   }
void Rectangle::Output( ) // 输出面积和周长
{
    cout << Area( ) << endl;
    cout << Len( ) << endl;
}
```

**程序员乙：类声明头文件（2.h）**

```
// 为长方形类 Rectangle 添加 2 个函数成员
class Rectangle   // 类声明部分
{
public:
    double a, b; // 数据成员：保存长度和宽度
    double Area( );  // 函数成员：计算面积
    double Len( );   // 函数成员：计算周长
    void Input( );   // 函数成员：输入长宽
    void Output( );  // 函数成员：输出结果
};
```

例 7-4 程序的代码说明如下。

**1）程序员乙继续完善长方形类 Rectangle 的功能**

程序员乙在例 7-3 长方形类 Rectangle 的基础上，通过添加函数成员 Input 实现了输入原始数据（即输入长方形的长和宽）的功能，再添加函数成员 Output 又实现了输出处理结果（即显示长方形的面积和周长）的功能。

**2）程序员甲使用功能完善后新的长方形类 Rectangle**

程序员甲编写主函数的目的是为了输入长方形长宽，然后计算其面积和周长。使用功能完善后新的长方形类 Rectangle，程序员甲在主函数中仅仅编写如下 3 条语句就实现了所需要的程序功能。

```
Rectangle   rect;                  // 定义一个长方形对象 rect
rect.Input( );                     // 调用长方形对象 rect 的函数成员 Input
rect.Output( );                    // 调用长方形对象 rect 的函数成员 Output
```

其中，调用函数成员 Input 的目的是输入长方形对象 rect 的长和宽。为什么调用函数成员 Input 就能输入长宽呢？因为程序员乙在定义 Input 函数成员时编写了如下的 cin 语句：

```
cin >> a >> b;
```

同理，调用函数成员 Output 就能输出长方形对象 rect 的面积和周长，因为程序员乙在定义 Output 函数成员时编写了如下的 cout 语句：

```
cout << Area( ) << endl;            // 调用函数成员 Area 计算面积，并通过 cout 显示出来
cout << Len( ) << endl;             // 调用函数成员 Len 计算周长，并通过 cout 显示出来
```

至此，程序员乙所编写的类 Rectangle 已完成了处理长方形数据所需的全部 4 项功能。可以看出，随着所承担功能的增多，程序员乙编写的代码越来越长，反过来程序员甲编写的代码则越来越短。程序员乙编写的长方形类 Rectangle 是能被重用的代码，其功能越强，重用的价值就越大。

仔细分析例 7-4 的程序，程序员乙编写的长方形类 Rectangle 总共包含了 6 个成员，分别是 2 个数据成员 a 和 b，以及 4 个函数成员 Area、Len、Input 和 Output。但程序员甲在使用 Rectangle 类时，只需要用 Input 和 Output 这两个成员就能完成所需要的程序功能，即输入长方形的长和宽，然后输出其面积和周长。换句话说，程序员甲在使用 Rectangle 类时不需要**直接访问**另外 4 个成员，即数据成员 a 和 b、函数成员 Area 和 Len。**请注意**：不需要直接访问的成员并不意味着是无用的成员，因为它们会被**间接访问**。例如，程序员甲在调用函数成员 Input 时会间接访问数据成员 a 和 b，在调用函数成员 Output 时会间接调用函数成员 Area 和 Len。

定义类的程序员可以将需要被外部直接访问的成员开放出来，同时将不需要被直接访问的成员隐藏起来，这就是面向对象程序设计中类的**封装**。

### 1. 类的封装

基于分类管理的思想，面向对象程序设计方法将程序中具有内在关联关系的变量和函数组合在一起形成类。类是一个整体，变量和函数是类的下属成员。类的**封装**有两层含义。

（1）**开放**。定义类时将必须被外部访问的成员开放出来，以保证类的功能可以被正常使用。

（2）**隐藏**。定义类时将不需要被外部访问的成员隐藏出来，以防止它们被误访问。被隐藏的成员只能在内部使用，被类里的其他成员访问，而在类的外部不能直接访问。

封装是一种常见的防护方法。例如设计电视机时要考虑电视机内部有很多电子元器件，需要用一个外壳将它们封装起来，这样可以防止用户误操作；同时将使用电视机所必须的操作（例如切换频道、调节音量等）通过一些按键开放出来，这样用户就可以使用这些按键来操作电视机。

C++语言中，封装是通过为类成员赋予不同的访问权限来实现的。访问权限有三种，它们分别是：

（1）**公有权限**（public）。被赋予公有权限的类成员是开放的，称为公有成员。

（2）**私有权限**（private）。被赋予私有权限的类成员将被隐藏，称为私有成员。

（3）**保护权限**（protected）。被赋予保护权限的类成员是半开放的，称为保护成员。

编写类的程序员通过设定访问权限将类成员封装起来，将需要访问的成员开放出来，不需访问的成员隐藏起来，这样可以避免误访问。访问权限是编写类的程序员为保证其他

程序员正确访问类成员所做的封装。

公有成员是封装后类提供给外界的操作接口。通常一个类必须有公有成员，否则这个类无法使用。程序员设计类时应根据功能要求合理设定成员权限，一方面要开放用户正常使用所必需的成员，另一方面要尽可能隐藏不被直接访问的成员。

### 2. 封装长方形类 Rectangle

程序员乙封装长方形类 Rectangle，就是在其类声明部分为各成员设定访问权限。在例 7-4 中，程序员乙将长方形类 Rectangle 的所有成员都设定为公有权限 public，即将它们都开放出来，这相当于没有封装。

下面程序员乙将根据例 7-4 中长方形类 Rectangle 的功能要求，重新修改各成员的访问权限。将需要被外部访问的 2 个成员 Input 和 Output 设定为公有权限 public，即将它们开放出来；将不需被外部直接访问的另外 4 个成员 a、b、Area 和 Len 设定为私有权限 private，即将它们隐藏起来。图 7-2 分别给出该长方形类 Rectangle 修改前后的封装示意图，其中类成员名前面的“+”表示公有权限（即对外开放的接口），“-”表示私有权限（被隐藏了）。

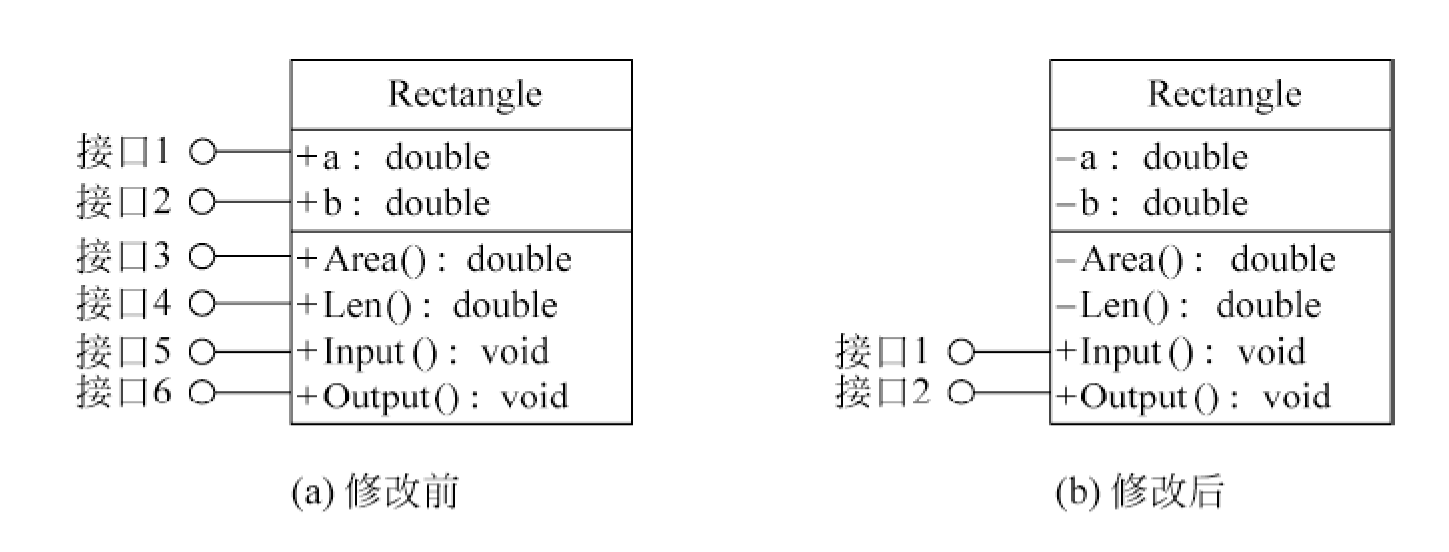



图 7-2　例 7-4 中长方形类 Rectangle 修改前后的封装示意图

重新设定访问权限后，例 7-4 中长方形类 Rectangle 声明部分的代码被修改如下：

```
class Rectangle                    // 在类声明部分设定各成员的访问权限
{
private:                           // 以下 4 个成员被设定为私有权限
        double a, b;               // 数据成员：保存长度和宽度
        double Area( );            // 函数成员：计算面积
        double Len( );             // 函数成员：计算周长
public:                            // 以下 2 个成员被设定为公有权限
        void Input( );             // 函数成员：输入长和宽
        void Output( );            // 函数成员：输出结果
};
```

访问权限只在类声明部分设定，与类实现部分的代码无关。重新设定长方形类 Rectangle 的访问权限，只需要修改其类声明部分的代码。

在程序员乙重新设定访问权限之后，程序员甲使用新的长方形类 Rectangle 定义对象，将只能访问对象中的公有成员。例如，

```
Rectangle   rect;                  // 定义一个长方形对象 rect
rect.Input( );                     // 调用公有的函数成员 Input，输入长方形的长和宽
rect.Output( );                    // 调用公有的函数成员 Output，显示长方形的面积和周长
```

可以看出，程序员甲使用新的长方形类 Rectangle 仍能够实现正常的程序功能。这说明程序员乙所设定的访问权限是合理的，他将程序员甲正常使用所必须的成员都开放出来了。

另一方面，在程序员乙重新设定访问权限之后，程序员甲将不能访问长方形类对象 rect 中的私有成员（或保护成员）。例如，程序员甲需要从键盘输入长方形对象 rect 的长和宽，这时他只能调用公有函数成员 Input 来实现，而不能用 cin 指令直接输入。例如，下面这条输入语句就存在语法错误，因为它要访问对象 rect 的私有成员 a、b。

```
cin >> rect.a >> rect.b;           // 访问私有成员，编译时编译器将提示语法错误
```

可以看出，虽然长方形对象 rect 包含私有成员 a、b，但是不能访问。程序员乙将数据成员 a、b 设为私有成员的目的是：迫使程序员甲只能通过公有的函数成员 Input 才能输入长宽数据。这么做有什么好处呢？请看下面经过改进的 Input 函数代码：

```
void Rectangle::Input( )           // 输入长方形的长度和宽度
{
    cout << "请输入长和宽：";   cin >> a >> b;
    while ( a < 0 || b < 0 )       // 数据合法性检查
    {   cout << "长宽值不能为负数，请重新输入：";   cin >> a >> b;   }
}
```

这个新的 Input 函数对所输入的长宽数据进行合法性检查，要求它们的数值不能是负数。强制通过这个 Input 函数来输入长宽数据，可以保证所输入的数值都是合法有效的。

本节通过具体的程序实例介绍了结构化程序设计到面向对象程序设计的演变过程，相信读者对这两种程序设计方法已经有了比较直观的认识。

自 1965 年提出结构化程序设计概念，再到 1989 年 ANSI 发布第一个 C 语言标准（ANSI C 或 C 89），结构化程序设计方法风靡全球。C 语言支持结构化程序设计方法，也曾是很长一段时间内全球使用最为广泛的计算机语言。但到了 1998 年，面向对象程序设计的 C++ 语言被 ISO 和 ANSI 两大标准化组织同时批准为国际标准（ISO/IEC14882 或 C++ 98），由此程序设计便迈入了面向对象的时代。面向对象程序设计在结构化程序设计的基础上，引入了**分类**（抽象）、**封装**、**继承**和**多态**的思想，将程序设计方法推向了一个新的高度。面向对象程序设计方法可以更好地组织和管理程序代码，也更便于重用代码。本章先介绍分类和封装，下一章再介绍继承和多态。

## 本节习题

1. 下列关于类的描述中，错误的是（　　）。
   A. 类可认为是一种高级数据类型　　B. 用类所定义出的变量称为对象
   C. 类包含数据成员和函数成员　　D. 类成员的访问权限有两种
2. 下列关于重用代码的描述中，错误的是（　　）。
   A. 函数是重用算法代码的语法形式

B. 结构体类型是重用数据代码的语法形式
C. 类是同时重用算法代码和数据代码的语法形式
D. 类是一种数据类型，因此只能重用数据代码

3. 关于程序开发过程中的程序员角色，下列哪种描述是错误的？（　　）
A. 一个程序员可以为其他程序员提供代码，即代码提供者
B. 一个程序员可以使用其他程序员提供的代码，即代码使用者
C. 一个程序员可以既是代码提供者，同时又是代码使用者
D. 一个程序员不能既是代码提供者，同时又是代码使用者

4. 关于程序设计方法，下列哪种描述是错误的？（　　）
A. 程序设计方法是研究如何对大型程序设计任务进行分解的方法
B. 结构化程序设计分解出的函数是一种算法零件
C. 结构化程序设计分解出的结构体类型是一种数据零件
D. 面向对象程序设计分解出的类是一种数据零件

5. 下列哪种思想不属于面向对象程序设计？（　　）
A. 抽象　　B. 封装　　C. 继承　　D. 模块化

## 7.2　面向对象程序的设计过程

程序员拿到一个具体的设计任务，该如何运用面向对象的方法进行程序设计呢？第 5.1.1 节曾以一个测算养鱼池工程总造价的设计任务为例，讲解了结构化程序设计方法的设计过程。本节我们继续以这个设计任务为例，讲解面向对象程序设计方法的设计过程。对于相同的任务但使用不同的设计方法，读者可以通过分析比较，深入体会两者的不同之处。

再回顾一下这个程序设计任务：公园计划修建一个长方形观赏鱼池，另外配套修建一大一小两个圆形蓄水池，分别存放清水和污水（参见图 5-1）。养鱼池和蓄水池的造价均为 10 元/$m^2$。请设计一个测算养鱼池工程总造价的计算机程序。

面向对象程序的设计过程，简单地可分为**分析**、**抽象**和**组装**等三个阶段。本节以统一建模语言（UML）来描述设计结果，然后再用 C++语言将设计结果编写成计算机程序。

### 7.2.1　分析

面向对象程序设计的第一阶段是需求分析。程序员经过需求调研可以知道，使用计算机测算养鱼池工程总造价的工作流程大致如下。

（1）**用户**启动**测算程序**，由计算机执行这个程序。

（2）程序等待用户输入原始数据，其中包括养鱼池的长和宽、清水池和污水池的半径。用户输入数据后，程序继续执行。

（3）程序要在计算机中分别创建**养鱼池**、**清水池**和**污水池**，计算并汇总其造价。

（4）显示汇总后的工程总造价。测算程序结束。

通常，程序员以**用例图**的形式来描述工作流程和功能需求，然后对工作流程进行分析，

从中提取出所有参与流程的**客观事物**，并以**顺序图**、**活动图**和**状态图**等形式描述各客观事物是如何参与工作流程的。例如，从测算养鱼池工程总造价的工作流程中共提取出用户、测算程序、养鱼池、清水池和污水池等 5 个客观事物。使用顺序图来描述它们如何参与测算养鱼池工程总造价的流程，如图 7-3 所示。

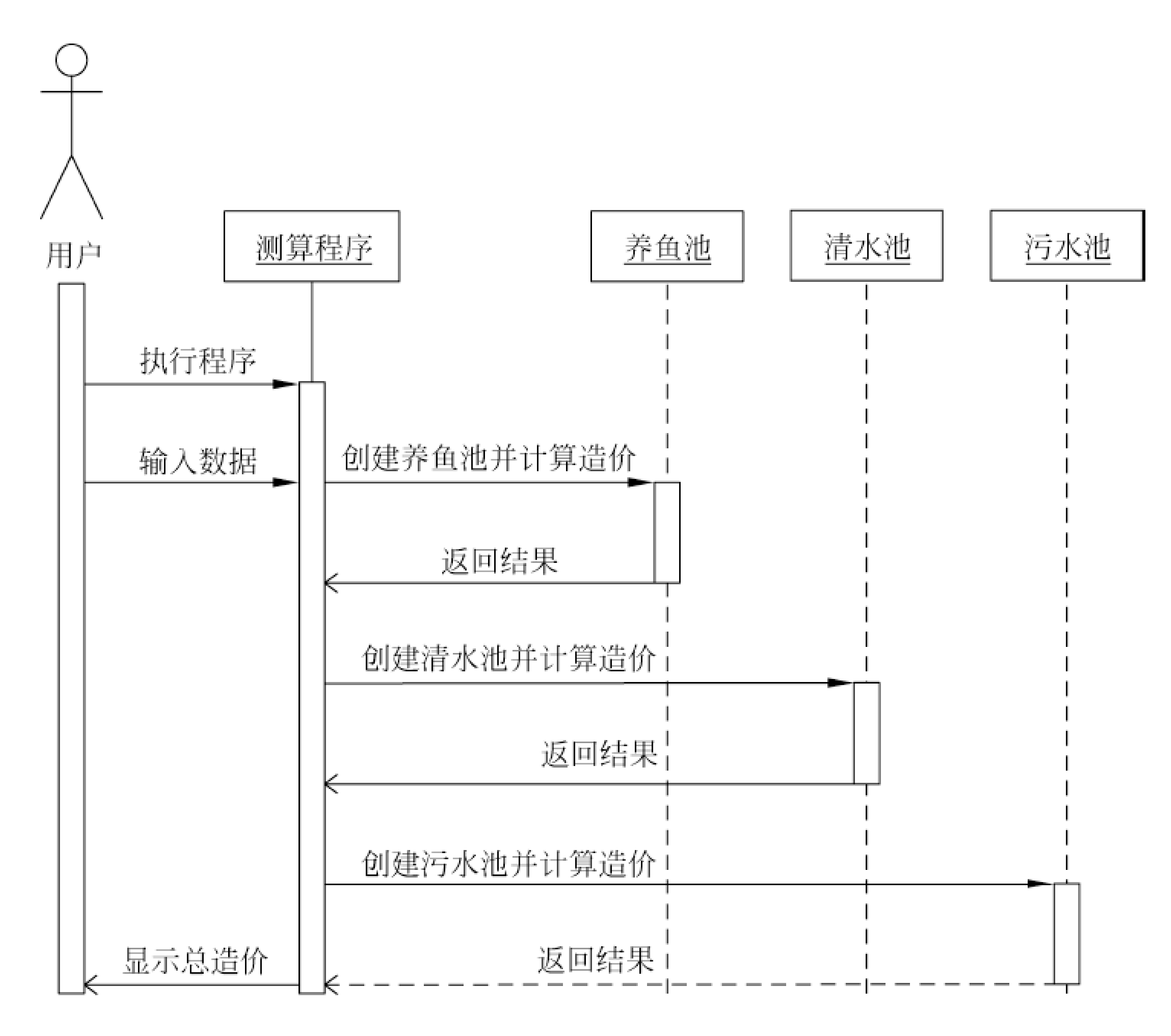



图 7-3 顺序图示例

图 7-3 中的客观事物“测算程序”是指一个程序设计任务，即测算养鱼池工程总造价的程序。在 C++语言中，就是要编写一个描述测算养鱼池工程总造价流程的主函数 main。主函数是与程序设计任务相关的，每个程序设计任务都要编写自己的主函数。

在测算养鱼池工程总造价程序中，还要对养鱼池、清水池和污水池这 3 个客观事物进行处理，求出它们的造价。下面问题的关键是，如何在程序中描述养鱼池、清水池和污水池这 3 个客观事物？如何处理它们？

使用结构化程序设计方法，可以将客观事物分解成数据和算法两部分。计算机程序使用变量来保存描述客观事物的数据，使用函数来描述处理客观事物的算法。例如在程序中描述和处理长方形养鱼池，程序员可以编写如下的代码：

```
double length, width;                    // 定义 2 个变量，分别保存长方形养鱼池的长宽数据
double RectCost(double a, double b)      // 定义 1 个函数，描述计算养鱼池造价的算法
{
    double cost ;
    cost = a * b * 10 ;                  // 造价＝长×宽×单价
    return cost ;
}
```

可以看出，计算机程序要处理客观世界中的事物，首先要对客观事物进行抽象，提炼出数据模型。程序设计对客观事物的处理实际上是对其数据模型的处理。7.2.2 节将介绍面向对象程序设计是如何将客观事物抽象成数据模型的。

## 7.2.2　抽象

面向对象程序设计的第二阶段是对客观事物进行抽象。计算机只能进行数值计算，要想让计算机来处理客观世界中的事物，首先需要为客观事物建立**数据模型**。数据模型应包含以下两方面的内容。

（1）**属性**（property）。为描述清楚客观事物，需要哪些数据？描述事物的数据被称为属性。例如在测算养鱼池工程总造价程序中，描述长方形养鱼池需要长和宽这两个属性。计算机程序通过定义**变量**来存储属性数据。变量就是数据模型中的属性。

（2）**方法**（method）。对客观事物要进行什么样的处理？描述事物处理的算法被称为方法。例如在测算养鱼池工程总造价程序中，对长方形养鱼池的处理就是计算其造价。计算机程序通过定义**函数**来描述算法。函数就是数据模型中的方法。

综上所述，长方形养鱼池的数据模型包含两个属性（即长和宽），以及一个计算造价的方法。在测算养鱼池工程总造价程序中，我们还需要处理圆形清水池和圆形污水池。凭直觉，这两个水池有点类似。是的，圆形清水池和圆形污水池具有相同的数据模型，它们都包含一个属性“半径”和一个“求圆形水池造价”的方法。

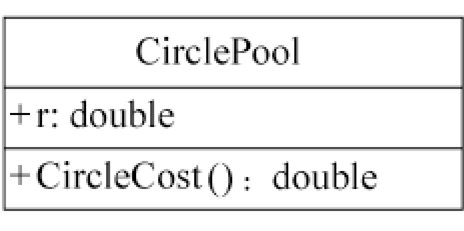
面向程序设计将客观世界中的事物称为一个个具体的**客观对象**。将具有相同数据模型的客观对象归纳成一类，这就是面向对象程序设计中的**分类**。对客观事物进行归纳，划分成不同的类，这是人类认识客观世界，解决实际问题常用的思维方法。分类就是抓住主要特征，忽略次要特征，将具有共性的事物划分成一类。分类的过程是一个不断抽象的过程，面向对象程序设计将分类称为**抽象**。

面向对象程序设计将抽象得到的数据模型称为**类**，其中包含属性成员和方法成员。每个类都代表了一组客观世界中具有共性的事物，它是这组客观事物抽象后得到的数据模型。进一步完善类的设计，合理设定各成员的访问权限，这就是类的**封装**。UML 用**类图**来描述类的设计结果。

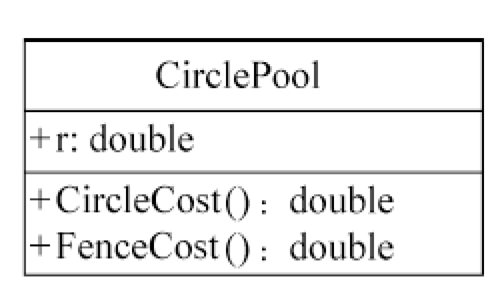
在测算养鱼池工程总造价程序中，长方形养鱼池被抽象成一个长方形鱼池类（命名为 RectPool），圆形清水池和圆形污水池被抽象成一个圆形水池类（命名为 CirclePool）。用类图描述上述设计结果，图 7-4(a)是长方形鱼池类 RectPool 的类图，图 7-4(b)是圆形水池类 CirclePool 的类图。

RectPool
+a: double
+b: double
+RectCost() : double

(a) RectPool的类图




(b) CirclePool的类图




(c) 有防护栏造价的CirclePool类图

图 7-4　类图示例

一个类到底要提炼多少属性和方法，这要依据程序的功能要求而定。假设公园修建的清水池和污水池还要增加防护栏，那么就需要为圆形水池类增加一个求防护栏造价的方法。图 7-4(c) 给出添加求防护栏造价方法 FenceCost 后圆形水池类 CirclePool 的类图。我们还可以继续为类添加功能，对类进行合理封装，优化类的设计，这样可以更加方便类的使用。

面向对象程序的分类设计过程是一个**“自底向上，逐步抽象”**的过程。从一个个具体的客观对象可抽象出小类，从小类可以抽象出更大的类。例如长方形水池类可进一步抽象出更宽泛的长方形类，圆形水池类也可以抽象出更宽泛的圆形类。长方形类和圆形类还可以再抽象出几何形状类。越往上，类就越宽泛、越抽象。

C++语言支持面向对象程序设计，用类（class）的语法形式来描述类图。定义类就是要描述清楚该类包含哪些数据成员（即属性成员）、函数成员（即方法成员）以及各成员的访问权限，定义时使用关键字 class。例如，程序员使用 C++语言来定义长方形鱼池类 RectPool 和圆形水池类 CirclePool，其示意代码如下。

（1）使用 C++语言定义图 7-4(a)的长方形鱼池类 RectPool。

```
class RectPool                  // 声明部分：声明类中有哪些成员以及各成员的权限
{
public:                         // 以下成员为公有权限
    double a, b;                // 声明数据成员：长度，宽度
    double RectCost( );         // 声明函数成员的原型：计算长方形鱼池造价
};
// 实现部分：函数成员 RectCost 的完整定义代码
double RectPool :: RectCost( )
{    return  (a*b *10);    }
```

（2）使用 C++语言定义图 7-4(b)的圆形水池类 CirclePool。

```
class CirclePool                // 声明部分：声明类中有哪些成员以及各成员的权限
{
public:                         // 以下成员为公有权限
    double r;                   // 声明数据成员：半径
    double CircleCost( );       // 声明函数成员的原型：计算圆形水池造价
};
// 实现部分：函数成员 CircleCost 的完整定义代码
double CirclePool :: CircleCost( )
{    return  (3.14 * r*r *10);    }
```

一个类的定义代码通常被分为**声明**和**实现**两部分。类声明部分在大括号中声明数据成员和函数成员，并指定各成员的访问权限。声明数据成员的语法形式类似于定义变量的语句，但声明时不能初始化。声明函数成员只是声明其原型，完整的函数定义代码被放在大括号外面的类实现部分，定义时需在函数名前加“类名::”进行限定，指明该函数成员是属于哪个类的。

类图相当于是**设计时**描述客观事物（即客观对象）数据模型的图纸，而类定义则是**编程时**描述该数据模型的 C++代码。

### 7.2.3　组装

在设计好类之后，面向对象程序设计进入最后的程序组装阶段。组装程序就是编写主函数，先生产程序零件，然后将它们组装在一起，最终形成完整的程序产品。

类相当于是描述客观事物（即客观对象）数据模型的图纸。可以按照图纸来生产产品，所生产出的产品称为是图纸的**实例**（instance）。在面向对象程序设计中，类被看作是一种由程序员定义的高级数据类型，用类所定义的变量称为**对象**（object）。使用类定义对象，这相当于是按照类图纸来生产对象，因此对象也被称为类的实例，定义对象被称为对类的实例化。例如，下面的定义对象语句使用圆形水池类 CirclePool 定义一个圆形清水池对象 cObj1 和一个圆形污水池对象 cObj2。

```
CirclePool   cObj1, cObj2;                    // 对象 cObj1 和 cObj2 是圆形水池类 CirclePool 的实例
```

计算机执行上述定义对象语句，将按照类图纸在内存中创建（即生产）对象 cObj1 和 cObj2，为它们分配内存。这种在内存中创建的对象被称为**内存对象**。图 7-5 以圆形清水池和圆形污水池为例，具体演示了面向对象程序设计从客观对象到内存对象的全过程。

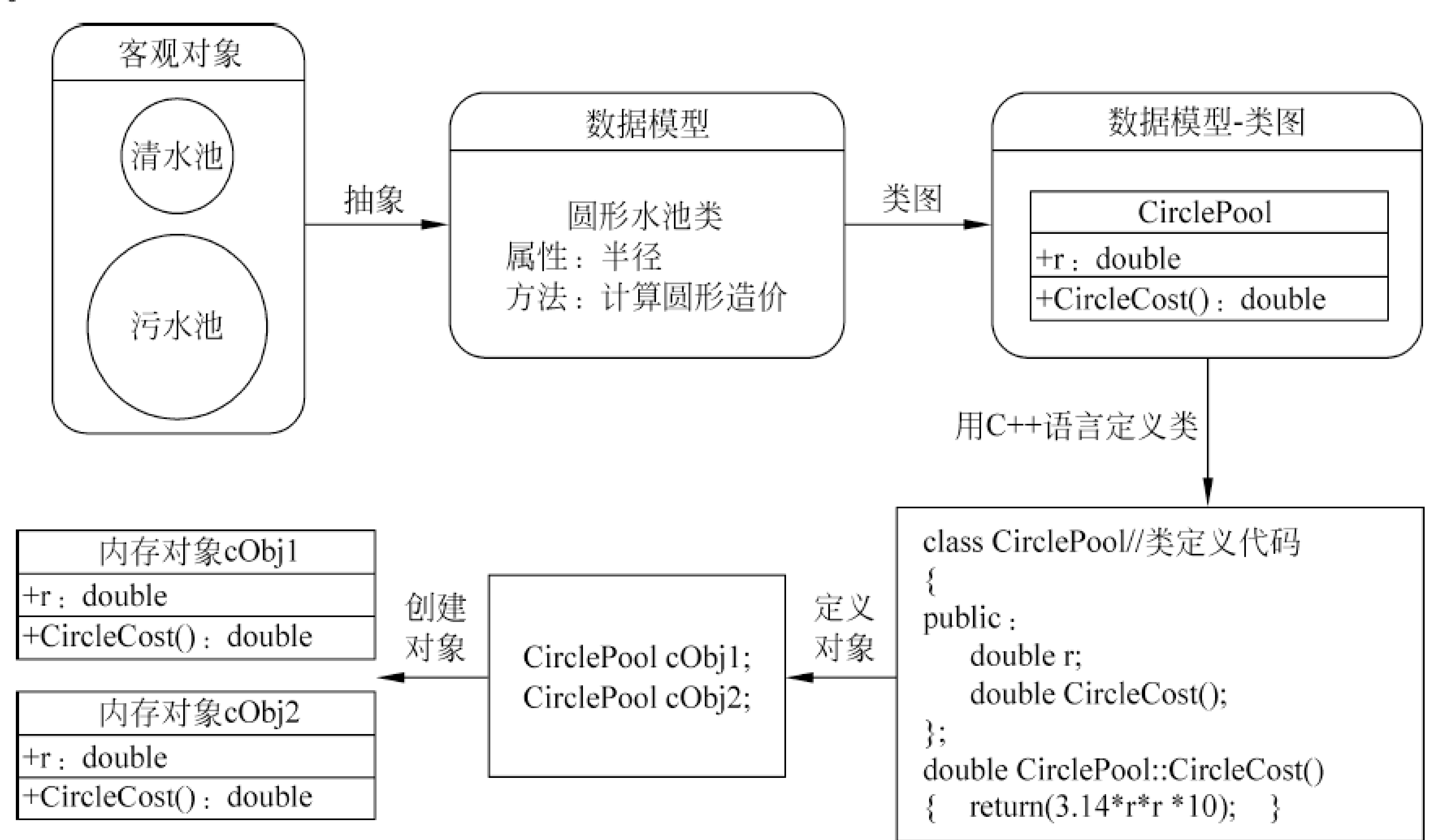



图 7-5　从客观对象到内存对象的设计过程

从图 7-5 可以看出，内存对象 cObj1 和 cObj2 实际上是客观对象圆形清水池和圆形污水池经过抽象后在内存中的表现形式。内存对象具有类所规定的数据成员、函数成员及访问权限。在理解了内存对象和客观对象的概念之后，请读者记住：后续章节提到程序中的**对象**通常指代的都是内存对象。

用类定义出对象之后，程序员可以访问对象的公有成员。公有成员是对象对外开放的接口。访问公有成员就是通过接口来操作内存中的对象，例如访问公有数据成员来读写对象的数据，或调用公有函数成员来处理对象。

```
cin >> cObj1.r >> cObj2.r;            // 输入清水池对象 cObj1 和污水池对象 cObj2 的半径
cout << cObj1.CircleCost( );          // 计算并显示清水池对象 cObj1 的造价
cout << cObj2.CircleCost( );          // 计算并显示污水池对象 cObj2 的造价
```

假设测算养鱼池工程总造价程序由甲、乙两位程序员分工协作，共同编写。程序员乙负责定义类，编写长方形鱼池类 RectPool 和圆形水池类 CirclePool 的定义代码。程序员甲负责编写主函数，在主函数中使用类定义对象，然后访问对象的公有成员，最终完成测算养鱼池工程总造价的功能。

程序员甲在使用程序员乙定义的类之前，必须对类进行**声明**，即“先声明，后使用”。类的声明形式与定义类时的声明部分相同，可以直接复制过来。但为了方便程序员甲，程序员乙将类声明部分单独提出来，另外保存在一个头文件（.h）中。这样程序员甲就可以通过插入头文件来简化类的声明。例 7-5 给出最终完整的测算养鱼池工程总造价的 C++程序代码。

**例 7-5　测算养鱼池工程总造价的 C++程序代码（面向对象程序设计方法）**

**程序员甲：主函数（1.cpp）**

```
#include <iostream>
using namespace std;

#include "2.h"   // 插入类声明头文件 2.h

int main( )
{
    double totalCost = 0;

    // 处理长方形养鱼池
    RectPool   rObj;   // 定义对象
    cin >> rObj.a >> rObj.b;   // 输入长宽值
    totalCost += rObj.RectCost( );   // 汇总造价
    // 处理清水池和污水池
    CirclePool   cObj1, cObj2;   // 定义对象
    cin >> cObj1.r >> cObj2.r;   // 输入半径
    totalCost += cObj1.CircleCost( );   // 汇总造价
    totalCost += cObj2.CircleCost( );   // 汇总造价

    cout << totalCost << endl;   // 显示总造价
    return 0;
}
```

**程序员乙：类实现程序文件（2.cpp）**

```
#include "2.h"   // 插入类声明头文件 2.h

// 实现部分：各函数成员的完整定义
double RectPool :: RectCost( )
{     return   (a*b * 10);     }

double CirclePool :: CircleCost( )
{     return   (3.14 * r*r * 10);     }
```

**程序员乙：类声明头文件（2.h）**

```
class RectPool   // 长方形养鱼池类声明部分
{
public:
    double a, b;   // 声明数据成员
    double RectCost( );   // 声明函数成员
};

class CirclePool   // 圆形水池类声明部分
{
public:
    double r;   // 声明数据成员
    double CircleCost( );   // 声明函数成员
};
```

从例 7-5 中程序员甲所编写的主函数代码可以看出，使用类的程序员在编写程序时应该先用类定义对象，然后再通过对象访问其公有成员，最终完成所需要的程序功能。

对照例 7-5 和第 5 章例 5-6 的程序代码，它们的功能相同，都是测算养鱼池工程的总造价。但因为采用了不同的程序设计方法，进而编写出了不同的程序代码。例 5-6 采用的是结构化程序设计方法，而例 7-5 采用的是面向对象程序设计方法。针对相同任务，程序员可以使用不同的程序设计方法。目前，面向对象程序设计方法是主流。

学习面向对象程序设计方法，必须从代码分类管理、数据类型、归纳抽象和代码重用等多个维度才能准确理解其思想内涵。本节通过测算养鱼池工程总造价程序的例子，简单介绍了面向对象程序的设计过程，相信读者已经认识了面向对象程序设计方法的概貌。如果想进一步了解面向对象程序设计方法，请阅读面向对象软件工程，或 UML 统一建模语言等方面的参考书。从下一节开始，我们将具体学习 C++语言中类和对象的语法细则。

### 本节习题

1. 下列关于类的描述中，错误的是（　　）。
   A. 类是描述客观事物的数据模型　　B. 可以用流程图来描述类的设计
   C. 类的数据成员也被称作属性　　D. 类的函数成员也被称作方法
2. 按照面向对象程序设计的观点，下列关于对象描述中错误的是（　　）。
   A. 客观世界中的事物被称作客观对象
   B. 类是描述客观对象的数据模型
   C. 程序中用类定义出的对象被称作内存对象
   D. 同类的多个内存对象可以包含不同的成员
3. 关于面向对象程序设计方法，下列哪种描述是错误的？（　　）
   A. 面向对象程序设计方法中的类是客观事物抽象后的数据模型
   B. 面向对象程序设计方法更便于代码分类管理
   C. 面向对象程序设计方法所设计出的类代码不能重用
   D. 面向对象程序设计方法是当今程序设计的主流方法
4. 假设编写一个教务管理系统，通过分析可抽象出若干个类，其中应不包括下列哪个类？（　　）
   A. 学生类　　B. 教师类　　C. 课程类　　D. 宿舍类
5. 如果将客观世界中的钟表抽象成一个钟表类，其中不应当包含下列哪个成员？（　　）
   A. 时、分、秒　　B. 功率　　C. 设置时间　　D. 显示时间

## 7.3　类与对象的语法细则

面向对象程序设计方法将程序中的数据元素和算法元素按其内在关联关系统一进行分类管理，这就形成了类。类是结构体类型的进一步扩展，它既可以包含变量，又可以包含函数。类是整体，变量和函数是类的下属成员，分别称为数据成员和函数成员。类是一种由程序员定义的高级数据类型，用类所定义的变量称为对象。

C++语言支持面向对象程序设计方法，为类与对象编程提供了完善的语法规则。

### 7.3.1　类的定义

在 C++语言中，定义一个类就是用 C++语言描述该类包含了哪些数据成员、函数成员以及各成员的访问权限。

C++语法：定义类

```
class 类名  // 类声明（declaration）部分
{
public :
    公有成员
protected :
    保护成员
private :
    私有成员
} ;
各函数成员的完整定义  // 类实现（implementation）部分
```

语法说明：

- class 是定义类的关键字。
- 类名需符合标识符的命名规则。
- 类成员有两种，分别是数据成员和函数成员。某些特殊的类可能只包含一种成员，比如只包含数据成员，或只包含函数成员。
- 访问权限有三种，分别是 public（公有权限）、protected（保护权限）和 private（私有权限）。
- 类定义代码分为两个部分。类声明部分在大括号中声明数据成员、函数成员，并指定各成员的访问权限。声明数据成员的语法形式类似于定义变量的语句，但声明时不能初始化。声明函数成员只是声明其原型，完整的函数定义代码放在大括号外面的类实现部分。在类实现部分定义函数成员时需在函数名前加“类名::”限定，指明该函数属于哪个类。
- 函数成员可以访问本类中任意位置的数据成员，或调用本类中任意位置的函数成员。类成员之间互相访问不需要“先定义，后访问”，或“先声明，后访问”，也不受权限约束。在类定义代码（包含声明和实现两部分）范围内，任何类成员都可以被本类的其他成员访问，类成员具有类作用域（class scope）。数据成员相当于类中的全局变量，函数成员相当于类中的外部函数。
- 每个类都是一个独立的类作用域。不同类作用域之间的标识符可以重名，即不同类的成员之间可以重名。

例如，将客观世界中的钟表抽象成一个钟表类 Clock，其中应当包含时、分、秒等三个属性，还应当包含设置时间和显示时间这两种方法。使用 C++语言定义钟表类 Clock，将属性定义成变量（即数据成员），将方法定义成函数（即函数成员），其示意代码如下。

```
class Clock                    // 钟表类 Clock 的声明部分
{
private :                      // 以下成员为 private 权限，即私有成员
    int hour;                  // 声明数据成员 hour：保存小时数。私有成员
    int minute;                // 声明数据成员 minute：保存分钟数。私有成员
    int second;                // 声明数据成员 second：保存秒数。私有成员
public :                       // 以下成员为 public 权限，即公有成员
    void Set( );               // 声明函数成员 Set：设置时间。公有成员
    void Show( );              // 声明函数成员 Show：显示时间。公有成员
} ;
// 以下为钟表类 Clock 的实现部分
```

```
void Clock :: Set( )                    // 钟表类 Clock 函数成员 Set 的完整定义代码
{
    cin >> hour >> minute >> second;        // 从键盘输入时、分、秒
}
void Clock :: Show( )                   // 钟表类 Clock 函数成员 Show 的完整定义代码
{
    cout << hour << ":" << minute << ":" << second;          // 显示格式：时:分:秒
}
```

### 1. 数据成员的语法细则

（1）数据成员也称为属性，是类中的变量，用于保存数据。
（2）数据成员的类型可以是基本数据类型，也可以是自定义数据类型。
（3）不同数据成员之间的类型可以相同，也可以不同。
（4）数据成员不能与其他成员重名。
（5）声明数据成员的语法形式类似于定义变量，所不同的是声明时不能初始化。

### 2. 函数成员的语法细则

（1）函数成员也称为方法，是类中的函数，其功能通常是对本类中的数据成员进行处理。

（2）函数成员可直接访问本类中的数据成员，数据成员相当于是类中的全局变量。函数成员可以直接调用本类中的其他函数成员。

（3）在类中声明函数成员时可以指定形参的默认值，即带默认形参值的函数。

（4）不同函数成员之间可以重名，即重载函数。两个函数的形参个数不同，或数据类型不同，那么这两个函数就可以重载。函数成员不能与数据成员重名。

（5）可以将函数成员定义成内联函数。在类实现部分定义函数成员时可以用关键字 inline 将其定义成内联函数，或者直接将该函数成员定义在类声明部分的大括号里。C++ 编译器默认将直接定义在类声明部分大括号里的函数成员当做内联函数处理。

例如，在钟表类 Clock 中引入重载函数、带默认形参值的函数和内联函数，修改后的定义代码如下。

```
class Clock                             // 钟表类 Clock 的声明部分
{
private :                               // 以下成员为 private 权限，即私有成员
    int hour, minute, second;           // 声明 3 个数据成员：时、分、秒。私有成员
public :                                // 以下成员为 public 权限，即公有成员
    void Set( );                        // 不带参数的函数成员 Set：从键盘输入时间
    void Set(int h, int m, int s=0);    // 带参数的函数成员 Set：用参数设定时间
    // 上述两个函数 Set 为重载函数，它们分别提供了两种不同的设置时间方法
    // 其中第二个 Set 函数是一个带默认形参值的函数，秒数 s 的默认值为 0
    void Show( )                        // 函数成员 Show 直接定义在类声明中，默认为内联函数
    {
        cout << hour << ":" << minute << ":" << second;          // 显示格式：时:分:秒
    }
} ;
// 以下为钟表类 Clock 的实现部分
```

```
void Clock :: Set( )                          // 函数成员 Set：从键盘输入时间
{
    cin >> hour >> minute >> second;          // 从键盘输入时、分、秒
}
void Clock :: Set(int h, int m, int s)        // 函数成员 Set：用参数设定时间
{
    hour = h;   minute = m;   second = s;    // 将参数分别赋值给对应的数据成员
}
```

#### 3. 访问权限的语法细则

（1）每个类成员都有并且只有一种访问权限。

（2）定义类时，不同权限成员可以按任意次序编排。为便于阅读，通常将相同权限的成员编排在一起，或将数据成员编排在一起，将函数成员编排在一起。

（3）关键字 public、protected 和 private 指定了其后续成员的访问权限，默认为 private。在类声明中，同一关键字可出现多次，也可以不出现（如果没有该访问权限的成员）。

（4）通常，一个类应包含公有权限的成员，否则该类没有对外的接口，无法使用。

公有成员是类提供给外界的操作接口。程序员设计类时应根据功能要求合理设定成员权限。一方面要开放用户正常使用所必需的成员，另一方面要尽可能隐藏不想被直接访问的成员。例如，钟表类 Clock 在设计时将数据成员“时、分、秒”设为私有权限（即隐藏起来），为此它另外提供了公有的 Set 和 Show 方法（即对外开放的接口）来间接设置或显示时、分、秒的数据。

钟表类 Clock 通过定义数据成员、函数成员及各成员的访问权限提供了钟表的功能，例如设置钟表时间，或显示钟表时间。其他程序员使用钟表类 Clock 定义钟表对象，然后调用对象的公有方法 Set 和 Show 就能轻松实现钟表的功能。

### 7.3.2 对象的定义与访问

和 int、double 等基本数据类型相比，类是一种程序员自己定义的新的数据类型，可称为**类类型**。使用类，通常指的是用类定义变量。用类所定义的变量称为是该类的一个对象（或实例）。使用类应该“先定义，再使用”，或“先声明，再使用”。本节以 7.3.1 节中修改后的钟表类 Clock 为例，具体讲解对象的定义和访问语法。

#### 1. 定义对象

定义对象和定义变量的语法形式基本相同，例如定义一个钟表类 Clock 的对象 obj1：

```
Clock   obj1;   // obj1 是钟表类 Clock 的一个对象（或实例）
```

类就像是一张图纸，计算机执行定义对象语句就是按照图纸在内存中创建一个程序实例，即内存对象。计算机会严格按照类定义创建对象，所创建的对象具有类所规定的数据成员、函数成员及访问权限。按照 Clock 类的定义，其所定义的钟表对象 obj1 将包含 3 个私有的数据成员（即变量 hour、minute 和 second），另外还有 3 个公有的函数成员（2 个重载函数成员 Set 和 1 个内联函数成员 Show），共 6 个下级成员。

### 2. 访问对象

定义好的对象可以访问。访问对象就是通过接口（即公有成员）操作内存中的对象，实现特定的程序功能，比如读写对象中的公有数据成员，或调用其中的公有函数成员。对象中的公有成员可以访问，非公有成员（私有成员、保护成员）不能访问，这就是对象中成员的**访问权限控制**。

访问对象中公有成员需使用**成员运算符**“.”，以“对象名.数据成员名”的形式访问对象中的变量，以“对象名.函数成员名(实参列表)”的形式调用对象中的函数。例如，定义钟表类 Clock 的对象 obj1 之后，可以访问其公有成员实现钟表的功能，即先设置钟表对象 obj1 时间，然后再显示其时间。

```
obj1.Set( );                    // 调用钟表对象 obj1 的公有函数成员 Set，输入时、分、秒数据
obj1.Show( );                   // 调用钟表对象 obj1 的公有函数成员 Show，显示其时间
```

可以用钟表类 Clock 定义多个对象。例如，再定义第二个钟表对象 obj2，然后访问其公有成员，这样就可以设置并显示第二个钟表 obj2 的时间。

```
Clock   obj2;
obj2.Set( 10, 30 );             // 将钟表对象 obj2 的时间设为 10 点 30 分（秒数默认为 0）
obj2.Show( );                   // 显示钟表对象 obj2 的时间，显示结果：10:30:0
```

钟表类 Clock 中有两个设置时间的函数成员 Set，它们是重载函数。在设置钟表对象 obj1 的时间时没有给实参，编译器根据形参实参匹配原则将自动调用第一个不带参数的重载函数 Set，该函数通过键盘来输入时间。而在设置第二个钟表对象 obj2 的时间时给出了实参，编译器根据形参实参匹配原则将自动调用另一个带参数的重载函数 Set，将 obj2 的时间设为 10 点 30 分（秒数默认为 0）。

### 3. 对象的内存分配

计算机执行定义对象语句时将为对象分配内存，在内存中创建一个对象实例。理论上，一个对象所占用的字节数等于其所有数据成员占用字节数的总和。例如执行如下的定义对象语句：

```
Clock   obj1, obj2;  // 定义两个钟表类 Clock 的对象 obj1 和 obj2
```

按照钟表类 Clock 的定义，每个钟表对象都包含 3 个 int 型数据成员（即时、分、秒），共 12 个字节。图 7-6 给出了钟表对象 obj1 和 obj2 的内存分配示意图。**注**：在实际应用中，为了提高对象的内存访问速度，C++编译器会采用“字节倍数对齐”技术，每个对象所占用的内存可能大于其成员占用字节数的总和。

| | | 各数据成员在钟表类Clock中的定义 |
|---|---|---|
| obj1 | 4 字节 | int hour; |
| | 4 字节 | int minute; |
| | 4 字节 | int second; |
| obj2 | 4 字节 | int hour; |
| | 4 字节 | int minute; |
| | 4 字节 | int second; |

图 7-6　钟表对象 obj1 和 obj2 的内存分配示意图

### 7.3.3 对象指针

可以通过地址访问基本数据类型的变量，也可以通过地址访问类类型的对象。定义类类型的对象指针保存某个对象的地址，然后就可以通过对象指针间接访问该对象及其下级成员。例如：

```
Clock   obj;          // 定义一个类 Clock 的对象 obj
Clock   *p;           // 定义一个类 Clock 的对象指针 p
p = &obj;             // 将对象 obj 的地址赋值给同类型的对象指针 p
```

下面通过对象指针 p 间接访问 obj 的数据成员 hour、minute 和 second，输入对象 obj 的时间。例如：

```
cin >> (*p).hour >> (*p).minute >> (*p).second;
```

不幸的是，这条输入语句存在语法错误，因为它间接访问了钟表对象 obj 的私有成员 hour、minute 和 second。虽然钟表对象 obj 包含私有成员 hour、minute 和 second，但是不能访问。

钟表类 Clock 在设计的时候将数据成员时、分、秒设为私有权限（即隐藏起来），因此它另外提供了两个公有的 Set 方法来设置时间。可以按如下形式来设置并显示对象 obj 的时间：

```
(*p).Set( );          // 从键盘输入时间，或用参数设置时间：(*p).Set( 10, 30 );
(*p).Show( );         // 显示对象 obj 的时间
```

当对象指针 p 指向对象 obj 时，*p 与 obj 等价。此时可以通过对象名 obj，也可以通过对象指针 p 来访问对象的下级成员。例如，表 7-1 中左右两边的访问形式是等价的。

**表 7-1 通过对象名或对象指针访问下级成员的形式**

| 通过对象名 obj 访问下级成员 | 通过对象指针 p 访问下级成员 |
| --- | --- |
| obj.hour | (*p).hour |
| obj.minute | (*p).minute |
| obj.second | (*p).second |
| obj.Set( ) | (*p).Set( ) |
| obj.Show( ) | (*p).Show( ) |

由于通过“(*p).”的形式访问对象成员比较烦琐，C++语言引入了一种新的更加直观的运算符“->”，称为**指向运算符**。通过指向运算符访问对象成员的语法形式是：

以“**对象指针名->数据成员名**”的形式访问对象中的数据成员。

以“**对象指针名->函数成员名(实参列表)**”的形式调用对象中的函数成员。

例如，使用指向运算符访问对象指针 p 所指向对象 obj 的下级成员：

```
p->hour、p->minute、p->second、p->Set( )、p->Set( 10, 30 )、p->Show( )
```

使用对象指针需注意以下三点：

（1）对象指针与被访问的对象应具有相同的类类型。

（2）对象指针需先赋值，即先指向被访问的对象，然后才能间接访问该对象及其成员。

（3）即使通过对象指针也只能访问对象的公有成员，不能访问非公有成员（即私有成员、保护成员）。

## 7.3.4　类与对象的编译原理

源程序中的 C++代码需要编译成等效的机器语言指令才能被计算机硬件识别和执行。例 7-6 给出一个完整的钟表类 Clock 的 C++演示程序。下面就这个例子来具体讲解类代码的编译原理以及对象的创建和访问过程。

**例 7-6　钟表类 Clock 的 C++演示程序**

```
#include <iostream>
using namespace std;

class Clock                          // 钟表类 Clock 的声明部分
{
private :                            // 以下成员为 private 权限，即私有成员
    int hour, minute, second;        // 声明 3 个私有数据成员：时、分、秒
public :                             // 以下成员为 public 权限，即公有成员
    void Set( );                     // 不带参数的函数成员 Set：从键盘输入时间
    void Set(int h, int m, int s=0);   // 带参数的函数成员 Set：用参数设定时间
    void Show( )                     // 函数成员 Show 直接定义在类声明中，默认为内联函数
    {
        cout << hour << ":" << minute << ":" << second << endl;     // 显示格式：时:分:秒
    }
} ;
// 以下为钟表类 Clock 的实现部分
void Clock :: Set( )                 // 函数成员 Set：从键盘输入时间
{
    cin >> hour >> minute >> second;              // 从键盘输入时、分、秒
}
void Clock :: Set(int h, int m, int s)            // 函数成员 Set：用参数设定时间
{
    hour = h;   minute = m;   second = s;         // 将参数分别赋值给对应的数据成员
}

int main( )
{
    Clock   obj1, obj2;                           // 定义两个钟表类 Clock 的对象 obj1 和 obj2
    // 设置并显示第一个钟表 obj1 的时间
    obj1.Set( );      obj1.Show( );
    // 设置并显示第二个钟表 obj2 的时间
    obj2.Set( 10, 30 );      obj2.Show( );
    return 0;
}
```

### 1. 类代码的编译

编译时，编译器将源程序中的类定义代码编译成等效的机器语言指令。当编译类中的函数成员时，编译器会对其定义代码做出某些调整，然后再进行编译。例如，钟表类 Clock 中的函数成员 Set（带参数）和 Show 在编译时编译器会做出如下调整。

| 函数成员 Set(int h, int m, int s)：调整前 | 函数成员 Set(int h, int m, int s)：调整后 |
|---|---|
| void Set( int h, int m, int s )<br>{<br>    hour = h;<br>    minute = m;<br>    second = s;<br>} | void Set( **Clock * const this**, int h, int m, int s )<br>{<br>    **this**->hour = h;<br>    **this**->minute = m;<br>    **this**->second = s;<br>} |
| **函数成员 Show( )：调整前** | **函数成员 Show( )：调整后** |
| void Show( )<br>{<br>    cout << hour << ":"<br>        << minute << ":"<br>        << second << endl;<br>} | void Show( **Clock * const this** )<br>{<br>    cout << **this**->hour << ":"<br>        << **this**->minute << ":"<br>        << **this**->second << endl;<br>} |

可以看出，编译器在编译类中函数成员时会对其定义代码做出如下两点调整：

（1）增加一个形参 this，该形参是一个本类的对象指针。例如，

```
Clock * const   this
```

对象指针 this 是一个 Clock 类型的指针常变量，在函数体中不能修改其指向。

（2）修改函数体中所有对本类成员的访问形式，在成员名之前加“this->”，即：

```
this->成员名
```

例如，this->hour、this->minute 和 this->second，这是一种通过对象指针 this 间接访问类成员的形式。

编译器为什么要对类中的函数成员做这样的调整呢？我们将在下面调用这些函数成员时再加以解释。执行例 7-6 所示的 C++程序时，操作系统将其可执行文件读入内存，建立一个程序副本。图 7-7(a)给出了该程序副本的示意图。

### 2. 对象的创建

程序从主函数 main 开始执行。当执行到其中的对象定义语句时，计算机为对象分配内存空间，即在内存中创建对象。一个对象所占用的字节数等于其所有数据成员占用字节数的总和。例 7-6 中的主函数定义了两个钟表类 Clock 的对象 obj1 和 obj2（代码第 28 行），图 7-7(b)给出这两个对象在内存中的分配示意图。

本章 7.3.2 小节曾提到，计算机会严格按照类定义创建对象，所创建的对象具有类所规定的数据成员、函数成员及访问权限。按照 Clock 类的定义，其所定义的钟表对象应当包含 3 个数据成员（即变量 hour、minute 和 second），另外还应当包含 3 个函数成员（2 个设

置时间函数 Set 和 1 个显示时间函数 Show），总共应当有 6 个成员。

细心的读者可能发现，图 7-7(b)中的对象 obj1 和 obj2 都只包含数据成员，但没有包含任何函数成员。换句话说，计算机创建对象时只会为数据成员分配内存。因此内存中的对象实际上只是一个或一组数据，**对象 ＝ 数据**。

| 操作系统<br>及已运行的应用程序 | | |
|---|---|---|
| int main( )<br>{ … } | | 代码区 |
| Set(Clock *const this );<br>Set(Clock *const this, … );<br>Show(Clock *const this); | 类<br>Clock | 代码区 |
| 栈剩余内存 | 自动存储区(栈) | |
| 系统空闲内存 | 堆 | |

(a) 加载后的程序副本

| 操作系统<br>及已运行的应用程序 | | |
|---|---|---|
| int main( )<br>{ … } | | 代码区 |
| Set(Clock *const this);<br>Set(Clock *const this, … );<br>Show(Clock *const this); | 类<br>Clock | 代码区 |
| obj1.hour | obj1 | 自动存储区(栈) |
| obj1.minute | obj1 | 自动存储区(栈) |
| obj1.second | obj1 | 自动存储区(栈) |
| obj2.hour | obj2 | 自动存储区(栈) |
| obj2.minute | obj2 | 自动存储区(栈) |
| obj2.second | obj2 | 自动存储区(栈) |
| 栈剩余内存 | | 自动存储区(栈) |
| 系统空闲内存 | 堆 | |

(b) 主函数执行时的程序副本

图 7-7　执行例 7-6 程序时的内存示意图

可又为什么能够调用到对象的函数成员，函数成员的调用又是如何实现的呢？例如例 7-6 中的代码第 32 行：

```
obj2.Set( 10, 30 );          // 调用对象 obj2 的函数成员 Set 来设置时间
obj2.Show( );                // 调用对象 obj2 的函数成员 Show 来显示时间
```

上述问题的奥秘在于编译器。编译器在编译调用对象函数成员的语句时会对调用代码做出某些调整。

### 3. 调用对象函数成员语句的编译

编译器在编译通过对象调用其函数成员的语句时会先对调用形式做出某些调整，然后再编译。例如：

```
obj2.Set( 10, 30 );          // 调用对象 obj2 的函数成员 Set 来设置时间
```

在编译时会被调整为：

```
Set( &obj2, 10, 30 );
```

这条语句的含义是：调用钟表类 Clock 的函数成员 Set，调用时取出对象 obj2 的内存

地址，将其作为实参传递给函数 Set 的第一个形参，即对象指针 this。函数 Set 再通过 this 指针间接访问对象 obj2 的数据成员，从而实现了设置对象 obj2 时间的功能，例如：

```
void Clock :: Set( Clock * const this, int h, int m, int s )
{
    this->hour = h;     this->minute = m;     this->second = s;
}
```

此时通过 this 指针所访问的成员 hour、minute 和 second 就都是对象 obj2 的成员。可以看出，调用哪个对象的函数成员，this 指针就指向哪个对象，此时通过 this 指针所访问的成员就都是该对象的成员。

编译器在编译类代码时会对函数成员的定义形式进行调整，这里又对函数成员的调用形式做调整。之所以做这两处调整，其目的就是让多个同类对象共用一份函数代码，降低内存占用。例如，无论定义多少个钟表类 Clock 的对象，图 7-7 的内存程序副本中都只会保存一份函数成员的代码。

从概念上讲，计算机严格按照类定义创建对象，同类的多个对象都应该具有类定义中所规定的数据成员和函数成员。例如，钟表对象 obj1 和 obj2 都有各自的时、分、秒数值，需要单独分配内存保存各自的数据。为对象 obj1 分配的内存包含 3 个数据成员，即 obj1. hour、obj1.minute 和 obj1.second，为对象 obj2 分配的内存也包含 3 个数据成员，即 obj2. hour、obj2.minute 和 obj2.second。同类的多个对象都包含各自的数据成员，每个对象所占用的内存空间都等于类中全部数据成员所需内存空间的总和。

但对所有钟表对象来说，它们设置时间的方法一样，显示时间的方法也一样。因此编译器在编译时，通过调整函数成员的定义代码及调用形式，巧妙实现了执行时程序中的多个同类对象共用函数，内存中只需要保存一份函数代码。对象指针 this 在这中间扮演了重要角色。

程序员在编写函数成员代码时，形参 this 是隐含的，在函数体中访问其他类成员也不需要添加 this 指针，这些都由编译器在编译时自动添加。程序员可以在访问类成员时显式添加 this 指针，也可以通过 this 指针获取当前对象的地址，或用 this 指针将当前对象地址作为实参继续传递给其他函数。

## 本节习题

1. 下列关于类定义语法的描述中，错误的是（　　）。
   A. 定义类时需使用关键字 class
   B. 类定义代码通常分为声明和实现两部分
   C. 完整的函数定义代码通常放在类实现部分
   D. 不同类的成员之间不能重名
2. 下列关于类中数据成员的描述，错误的是（　　）。
   A. 数据成员用于保存数据
   B. 数据成员的类型只能是基本数据类型
   C. 类中的数据成员之间不能重名
   D. 声明数据成员不能初始化

3. 下列关于类中函数成员的描述，错误的是（　　）。
   A. 函数成员的功能通常是对本类中数据成员进行处理
   B. 函数成员访问本类中数据成员时需先定义，后访问
   C. 类中的函数成员之间可以重名，即重载函数
   D. 函数成员的完整定义代码可以放在声明部分，此时被当作内联函数来处理
4. 类成员的访问权限不包括下列哪种权限？（　　）
   A. public　　B. private　　C. protected　　D. inline
5. 下列关于对象的描述中，错误的是（　　）。
   A. 对象是用类定义的变量，也可称为是类的实例
   B. 理论上，一个对象所占的内存空间等于其类中所有数据成员所占内存的总和
   C. 一个对象只属于某一个类
   D. 一个类只能定义一个对象
6. 下列关于对象的描述中，错误的是（　　）。
   A. 对象包含哪些成员是由其类定义决定的
   B. 对象的数据成员用于保存数据，通过“对象名.数据成员名”进行访问
   C. 对象的函数成员用于处理数据，通过“对象名.函数成员名（）”进行调用
   D. 可以访问或调用对象中的所有成员
7. 已定义一个圆形类 Circle：

```
class Circle
{
private:    double r;
public:
        void SetR(double x)   {   r = x;   }
        double GetArea( )     {   return 3.14*r*r;   }
};
```

   下列计算圆面积的代码中正确的是（　　）。
   A. Circle c;　　c.r = 10.5;　　cout << 3.14*c.r*c.r;
   B. Circle c;　　c.SetR(10.5);　　cout << c.GetArea( );
   C. Circle c;　　cout << c.GetArea( );
   D. Circle c;　　SetR(10.5);　　cout << GetArea( );
8. 已定义一个圆形类 Circle：

```
class Circle
{
private:    double r;
public:
        void SetR(double x)   {   r = x;   }
        double GetArea( )     {   return 3.14*r*r;   }
};
```

   下列计算圆面积的代码中正确的是（　　）。
   A. Circle c, *p = &c;　　p.SetR(10.5);　　p.GetArea( );

B. Circle c, *p = c;　　p->SetR(10.5);　　p->GetArea( );
C. Circle c, *p = &c;　　p->SetR(10.5);　　p->GetArea( );
D. Circle c, *p = &c;　　p->GetArea( );

## 7.4 对象的构造与析构

程序执行过程中，变量从内存分配到释放这个时间段称为该变量在内存中的**生存期**。不同类型变量的分配方式不同，在内存中具有不同的生存期。全局变量和静态变量是**静态分配**的，它们在程序加载后立即被分配内存，运行过程中一直占用内存，直到程序执行结束退出时才被释放。局部变量是**自动分配**的，在执行到其定义语句时被分配内存，到其所在代码块执行结束即被释放。**动态分配**则是由程序员使用 new 和 delete 运算符自行决定内存何时分配，何时释放。

和变量一样，对象也有**全局对象**、**静态对象**和**局部对象**之分，也可以动态分配。程序执行时，对象也会经历从分配内存到释放内存的过程，即对象具有生存期。对象的内存分配方式不同，所具有的生存期也不同。对象的内存分配方式、生存期与变量完全一样。

程序执行过程中，计算机创建对象，为对象分配内存空间，我们称对象在内存中诞生了。当对象生存期结束时，计算机销毁对象，释放其内存空间，我们称对象死亡了。创建对象的过程称为对象的**构造**（construction），销毁对象的过程称为对象的**析构**（destruction）。

类就像是一张图纸，构造对象就是按照图纸在内存中创建一个内存对象。C++语言允许程序员参与对象的构造过程，实现对象的初始化，例如初始化对象的各个数据成员（相当于对象的出厂设置），或者构造对象时申请额外的内存等。同样，销毁对象时可能也需要程序员参与，例如释放额外申请的内存。

与基本数据类型的变量相比，类类型的对象可以包含多个数据成员，其构造和析构过程更复杂，涉及的内容更多。为此，面向对象程序设计需要定义专门的函数来实现构造与析构的功能，这就是类的构造函数与析构函数。

### 7.4.1 构造函数

C++语言允许程序员在类定义中添加构造函数，参与对象的构造过程。执行定义对象语句时，计算机将自动调用对象所属类的构造函数，实现对象的初始化。构造函数是类中的一种特殊函数成员。定义构造函数时应遵守以下几点特殊的语法细则：

（1）构造函数名必须与类名相同。

（2）构造函数由计算机自动调用，程序员不能直接调用。

（3）构造函数通过形参传递初始值（可指定默认形参值），实现对新建对象数据成员的初始化。

（4）构造函数可以重载，即定义多个同名的构造函数，这样可提供多种形式的对象初始化方法。

（5）构造函数可以定义成内联函数。

（6）构造函数没有返回值，定义时不能写函数类型，写 void 也不行。

（7）构造函数通常是类外调用，其访问权限应设为 public 或 protected，不能设为 private。

（8）一个类如果没有定义构造函数，编译器在编译时将自动添加一个空的构造函数，称为**默认构造函数**，其语法形式为：

```
类名( )  { }
```

构造函数的主要用途有初始化对象的数据成员，显示对象的构造过程，或者是在构造对象时申请额外内存等。

### 1. 初始化对象的数据成员

例如，使用例 7-6 中的钟表类 Clock 定义对象时可以初始化对象的数据成员，为它们赋以不同的值，这样就能定义出具有不同时、分、秒初始值的钟表对象。这些时、分、秒的初始值就相当于是钟表的出厂设置。

初始化对象的方法是先在类中添加构造函数，然后在定义对象时给出初始值。例如先在类 Clock 的定义代码中添加如下两个重载构造函数：

```
Clock :: Clock( int p1, int p2, int p3 )           // 带形参的构造函数，形参用于接收初始值
{
    // 在函数体中将形参接收到的时、分、秒初始值分别赋值给对应的数据成员
    hour = p1;   minute = p2;   second = p3;
}
Clock :: Clock( )                                  // 不带形参的构造函数，与上一个函数同名（即重载函数）
{
    // 因为未接收初始值，默认将时、分、秒都设置为 0（或任何其他数值）
    hour = 0;   minute = 0;   second = 0;
}
```

上述构造函数定义中，第一个“Clock”是类名，第二个“Clock”是构造函数名。

定义 Clock 类的钟表对象时可以给出时、分、秒的初始值。计算机执行定义对象语句时会自动调用构造函数，完成设置对象初始时间的功能。例如：

```
Clock obj1( 10, 30, 50 );   // 根据实参—形参匹配原则，自动调用带形参的构造函数
                            // 将对象 obj1 的初始时间设为 10 点 30 分 50 秒
Clock obj2;                 // 未给出实参，根据实参—形参匹配原则，自动调用不带形参的构造函数
                            // 将对象 obj2 的初始时间设为 0 点 0 分 0 秒
```

可以将上述两个重载函数改写成一个带默认形参值的函数，所完成的功能完全一样，但代码更加简洁。

```
Clock :: Clock( int p1 = 0, int p2 = 0, int p3 = 0 )      // 带默认形参值的构造函数
{
    // 如果定义对象时给出初始值，则形参接收初始值，否则使用默认值 0
    hour = p1;   minute = p2;   second = p3;           // 将初始值或默认值赋值给数据成员
}
```

**总结一下：**使用类定义对象，每定义一个对象就会调用一次类的构造函数，定义多少个对象就会调用多少次构造函数。需要注意的是，因为定义对象指针或对象引用并不会创建对象，因此计算机执行定义对象指针或对象引用语句时不会调用构造函数。

C++语言在定义基本数据类型变量时可以用赋值运算符“=”进行初始化，实际上也可以用面向对象的风格来初始化变量，例如：

```
int   x = 10,   y = 20;              // 传统形式：用“=”初始化变量
int   x( 10 ),   y( 20 );            // 面向对象的风格：用一对小括号( )来初始化变量
```

### 2. 用一个已存在的对象初始化当前新建对象

在类 Clock 的定义代码中再添加一个如下的新构造函数：

```
Clock :: Clock( Clock &rObj )      // 形参 rObj 为一个本类对象的引用，引用已经存在的对象
{
    // 将 rObj 所引用实参对象的数据成员一一对应地复制给当前对象的数据成员
    hour = rObj.hour;   minute = rObj.minute;   second = rObj.second;
}
```

该构造函数称为类 Clock 的拷贝构造函数。**拷贝构造函数**接收一个已存在的本类对象引用，然后将该对象的数据成员一一对应地复制给当前对象的数据成员，实现用一个已经定义的对象来初始化当前新建对象的功能。例如：

```
Clock obj1( 10, 30, 50 );   // 根据实参—形参匹配原则，自动调用带形参的构造函数
                            // 将对象 obj1 的初始时间设为 10 点 30 分 50 秒
Clock obj2( obj1 );         // 定义 obj2 时用 obj1 来初始化，将 obj2 也设为 10 点 30 分 50 秒
                            // 根据实参—形参匹配原则，自动调用拷贝构造函数
```

用一个已经定义的对象来初始化当前新建对象，这类似于用一个已经定义的变量来初始化当前新变量。例如：

```
int x = 10;
int y = x;          // 定义的新变量 y 用已经定义的变量 x 来初始化，初始化后 y 的值也为 10
```

可以看出，在类中每添加一个构造函数，就能为该类对象增加一种初始化方法。如果程序员没有为类定义拷贝构造函数，则编译器在编译时也会自动添加一个**默认拷贝构造函数**，其功能就是把实参对象的数据成员一一对应地复制给当前新建对象的数据成员。在定义拷贝构造函数时请注意一个语法细节，拷贝构造函数的形参只有一个，且必须是本类的引用。

### 3. 显示对象的构造过程

可以在构造函数的函数体中插入一些 cout 指令，这样可以让程序员在调试过程中实时观察到对象的构造过程，便于检查程序代码中的错误。例如，为类 Clock 中的三个构造函数添加如下的 cout 语句，用于显示当前哪个构造函数被调用了。

```
Clock :: Clock( )                        // 不带形参的构造函数
{
```

```
        hour = 0;    minute = 0;    second = 0;
        cout << "Clock( ) called." << endl;
}
Clock :: Clock( int p1, int p2, int p3 )            // 带形参的构造函数
{
        hour = p1;    minute = p2;    second = p3;
        cout << "Clock(int p1, int p2, int p3) called." << endl;
}
Clock :: Clock( Clock &rObj )                       // 拷贝构造函数
{
        hour = rObj.hour;    minute = rObj.minute;    second = rObj.second;
        cout << "Clock( Clock &rObj ) called." << endl;
}
```

这时执行下列定义对象语句，在自动调用构造函数时会执行其中的 cout 语句，这样就能在显示器上实时看到哪个构造函数被调用了。

```
Clock obj1;                   // 调用不带形参的构造函数，将显示 Clock( ) called.
Clock obj2(10, 30, 50);       // 调用带形参的构造函数，将显示 Clock(int p1, int p2, int p3) called.
Clock obj3( obj2 );           // 调用拷贝构造函数，将显示 Clock( Clock &rObj ) called.
```

以上提示信息显示出了对象的构造过程，这为程序员检查程序代码错误提供了线索。

#### 4. 构造对象时申请额外内存

下面以一个关于学生信息的类 Student 为例来讲解，什么情况下构造对象要申请额外的内存，以及申请额外内存时应如何编写构造函数。

```
class Student                          // 定义一个学生信息类 Student
{
public:
        char Name[9], ID[11];          // 保存姓名和学号的字符数组 Name、ID
        int Age;                       // 保存年龄的数据成员 Age
        double Score;                  // 保存成绩的数据成员 Score
        Student(char *pName, char *pID, int iniAge, double iniScore)    // 构造函数
        {
                strcpy(Name, pName);    strcpy(ID, pID);       // 初始化姓名、学号
                Age = iniAge;    Score = iniScore;             // 初始化年龄、成绩
        }
        // 其他函数成员省略
};
```

假设有某个同学张三，学号 1400500001，年龄 19 岁，成绩 95 分。定义一个保存该同学信息的对象 obj 时可以这样来初始化：

```
Student    obj( "张三", "1400500001", 19, 95 );
```

如果需要在类 Student 中增加一个备注信息 Memo，备注信息可有可无，有长有短，该如何定义数据成员呢？可以先在类中添加一个字符型指针 Memo，用于指向备注信息。

```
char *Memo;                            // 指向备注信息的字符指针 Memo
```

然后在构造函数中按照实际备注信息的长度来动态分配内存（即申请额外的内存）。

```
Student(char *pName, char *pID, int iniAge, double iniScore, char *pMemo)
{
    strcpy(Name, pName);      strcpy(ID, pID);      // 初始化姓名、学号
    Age = iniAge;      Score = iniScore;            // 初始化年龄、成绩
    // 下面按照实际备注信息 pMemo 的长度来动态分配内存
    int len = strlen( pMemo );                      // 计算实际传递来的备注信息长度
    if (len <= 0)      Memo = 0;                    // 没有备注信息。0 表示空指针
    else
    {
        Memo = new char[len +1];                    // 按照实际长度分配内存（需加一个结束符'\0'）
        strcpy(Memo, pMemo);                        // 初始化备注信息
    }
}
```

这样在定义类 Student 对象时就可以这样来初始化：

```
Student   obj( "张三", "1400500001", 19, 95, "" );                  // 无备注信息
```

或

```
Student   obj( "张三", "1400500001", 19, 95, "市级三好学生" );      // 有备注信息
```

合理使用动态内存分配技术可以有效降低内存的使用量。动态分配的内存在使用完之后需要由程序员编写 delete 语句来释放。在构造函数中动态分配的内存需要程序员编写析构函数，在析构函数中用 delete 语句来释放。

## 7.4.2 析构函数

当对象生存期结束时，计算机销毁对象，释放其内存空间，这个过程就是对象的析构。C++语言允许程序员在类定义中添加析构函数，参与对象的析构过程。计算机在销毁对象时将自动调用对象所属类的析构函数。与构造函数一样，析构函数也是类中的一种特殊函数成员。定义析构函数时应遵守以下几点特殊的语法细则：

（1）析构函数名必须为“~类名”。

（2）析构函数由计算机自动调用，程序员不能直接调用。

（3）析构函数没有形参。

（4）析构函数没有返回值，定义时不能写函数类型，写 void 也不行。

（5）一个类只能有一个析构函数。

（6）析构函数通常是类外调用，其访问权限应设为public或protected，不能设为private。

（7）一个类如果没有定义析构函数，编译器在编译时将自动添加一个空的析构函数，称为**默认析构函数**，其语法形式为：

```
~类名( )   {   }
```

析构函数的常见功能主要有清理内存，设置与当前对象相关的系统状态等。例如之前关于学生信息的类 Student，当增加备注信息 Memo 后，构造函数按照实际备注信息长度来

动态分配内存。在这种情况下，程序员必须要为类 Student 编写析构函数，用 delete 语句释放这些动态分配的内存。例如，

```
~Student( )
{
    if (Memo != 0)   delete [ ]Memo;        // 释放动态分配的内存
}
```

计算机在销毁对象时，每销毁一个对象就会调用一次类的析构函数，销毁多少个对象就会调用多少次析构函数。

### 7.4.3　拷贝构造函数中的深拷贝与浅拷贝

综合前述增加备注信息 Memo 后的构造函数和析构函数，例 7-7 给出一个学生信息类 Student 的 C++演示程序。

例 7-7　一个学生信息类 Student 的 C++演示程序

```
#include <string.h>

class Student                        // 定义一个学生信息类 Student
{
public:
    char Name[9], ID[11];            // 保存姓名和学号的字符数组 Name、ID
    int Age;                         // 保存年龄的数据成员 Age
    double Score;                    // 保存成绩的数据成员 Score
    char *Memo;                      // 保存备注信息的 char 型指针 Memo
    Student(char *pName, char *pID, int iniAge, double iniScore, char *pMemo)   // 构造函数
    {
        strcpy(Name, pName);     strcpy(ID, pID);       // 初始化姓名、学号
        Age = iniAge;     Score = iniScore;             // 初始化年龄、成绩
        int len = strlen( pMemo );                 // 计算实际传递来的备注信息长度
        if (len <= 0)     Memo = 0;                // 没有备注信息。0 表示空指针
        else                                       // 有备注信息
        {
            Memo = new char[len +1];   // 按照实际长度分配内存（需加一个结束符'\0'）
            strcpy(Memo, pMemo);                   // 初始化备注信息
        }
    }
    ~Student( )                                    // 析构函数
    {
        if (Memo != 0)   delete [ ]Memo;           // 释放动态分配的内存
    }
    // 其他函数成员省略
};

int main( )
```

```
30 {
31     Student  obj1( "张三",  "1400500001",  19,  95,  "市级三好学生" );  // 有备注信息
32     Student  obj2( obj1 );                        // 将自动调用拷贝构造函数
33     // 以下代码省略
34 }
```

例 7-7 中的类 Student 没有定义拷贝构造函数，编译器在编译时将自动添加一个默认拷贝构造函数，其语法形式如下：

```
Student( Student  &obj )
{
    strcpy(Name, obj.Name);    strcpy(ID, obj.ID);         // 复制姓名、学号
    Age = obj.Age;     Score = obj.Score;                 // 复制年龄、成绩
    Memo = obj.Memo;                          // 复制备注信息指针，注：并没有再分配内存
}
```

计算机在执行例 7-7 中代码第 32 行定义对象 obj2 的语句时将自动调用这个默认拷贝构造函数。其功能是把形参 obj 所引用实参对象 obj1 的数据成员一一对应地复制给新建对象 obj2 的数据成员。在复制备注信息 Memo 时没有再分配内存，只是简单做了指针赋值。赋值后 obj2.Memo 和 obj1.Memo 相等，即这两个指针变量指向了同一块内存单元，这种复制指针变量的形式被称为**浅拷贝**，如图 7-8(a)所示。

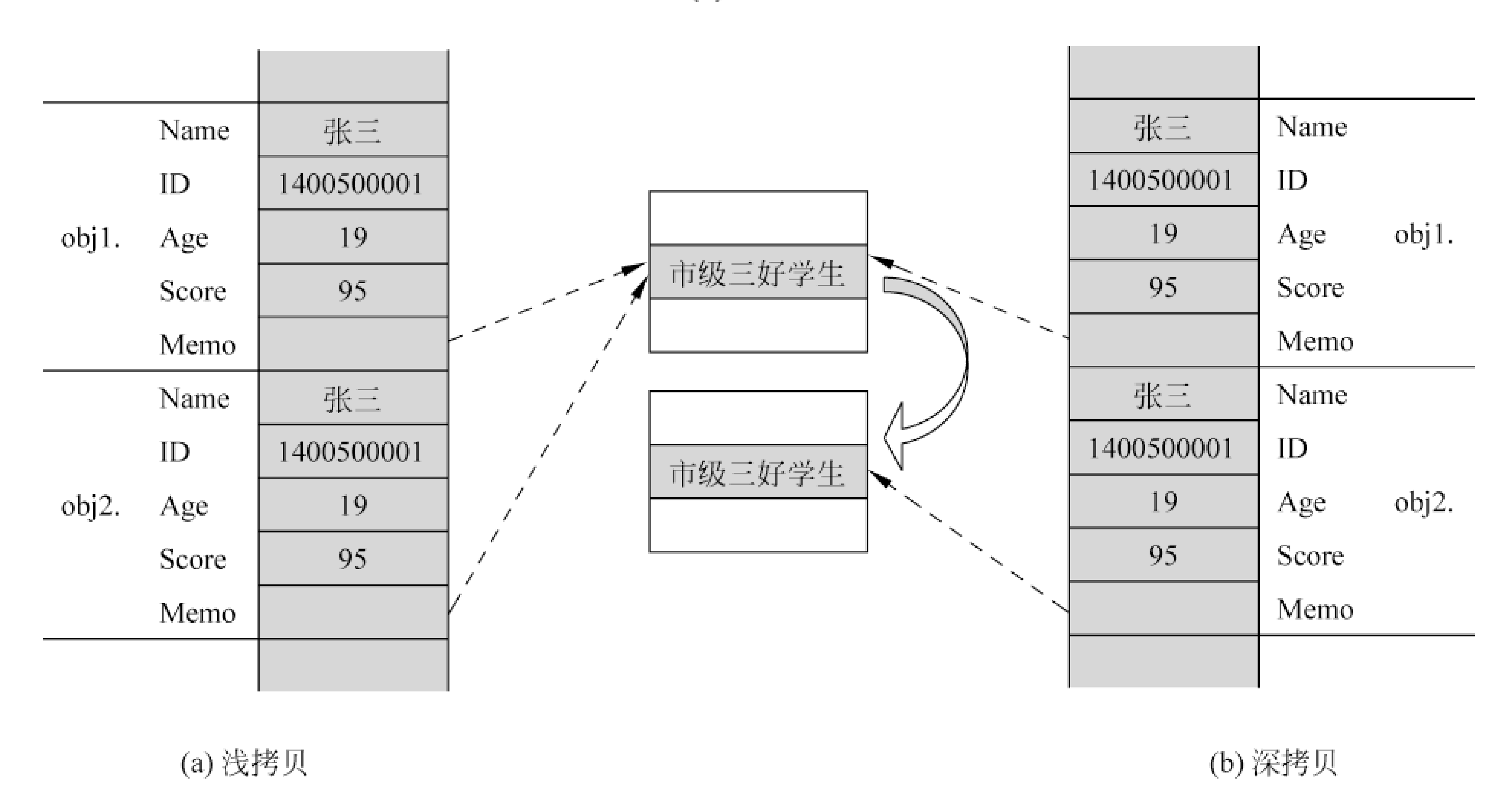



图 7-8　指针变量 Memo 的浅拷贝与深拷贝

程序员可以不让编译器自动添加默认拷贝构造函数，而是自己编写一个如下的拷贝构造函数：

```
Student( Student  &obj )
{
    strcpy(Name, obj.Name);    strcpy(ID, obj.ID);         // 复制姓名、学号
    Age = obj.Age;     Score = obj.Score;                 // 复制年龄、成绩
    int len = strlen( obj.Memo );                        // 计算 obj 所引用对象的备注信息长度
```

```
    if (len <= 0)       Memo = 0;                           // 没有备注信息。0 表示空指针
    else
    {
        Memo = new char[len +1];   // 按照实际长度再另外分配内存（需加 1 个结束符'\0'）
        strcpy(Memo, obj.Memo);    // 复制 obj 所引用对象的备注信息
    }
}
```

该拷贝构造函数在复制备注信息 Memo 时先另外分配内存，然后再复制内容，这种复制指针变量的形式被称为**深拷贝**。这时再执行例 7-7 中代码第 32 行定义对象 obj2 的语句时将自动调用这个新的拷贝构造函数。新建对象 obj2 将另外分配内存来保存备注信息，obj2.Memo 和 obj1.Memo 分别指向了不同的内存单元，见图 7-8(b)。

如果构造函数中动态分配了的内存，那么就需要编写析构函数，用 delete 语句释放这些动态分配的内存；同时还需要编写拷贝构造函数进行深拷贝，即为新建对象动态再分配同样多的内存，然后复制内容。

本节最后简单总结一下面向对象程序设计中类与对象编程的主要内容。

（1）**定义类**。程序员在定义一个类的时候要考虑 5 大要素，即数据成员、函数成员、各成员的访问权限、构造函数和析构函数。

（2）**定义对象**。将类当作一种自定义数据类型来定义变量，所定义的变量就称为对象。计算机执行定义对象语句就是严格按照类所描述的 5 大要素在内存中创建该类的对象，所创建的对象具有类所规定的数据成员、函数成员及访问权限。

（3）**访问对象**。访问对象就是通过接口（即公有成员）操作内存中的对象，实现特定的程序功能，比如读写对象中的公有数据成员或调用其中的公有函数成员。

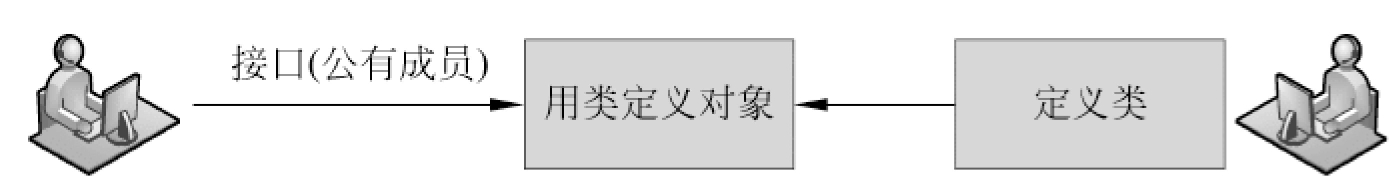
### 7.4.4　类与对象编程举例

类与对象编程主要包括三部分内容，即先定义类，然后用类定义对象，最后是访问对象的公有成员以实现特定的程序功能。本节通过一个程序实例来具体讲解类与对象编程的思路和方法。

**问题举例：**使用面向对象程序设计方法编写一个模拟银行存款账户管理的 C++程序。

**编程思路：**为了编写程序，程序员首先应分析清楚该程序涉及哪些数据，需要对这些数据进行什么处理。银行存款账户应包含账号、账户名和存款金额等数据。为简单起见，省略存款利率和存款日期等数据。管理银行存款账户应能够进行开户、存款、取款和查询余额等操作。使用面向对象程序设计方法，可以先定义一个银行账户类 Account，然后用类 Account 来定义账户对象 obj（即开户），通过访问对象 obj 的公有成员来实现存款、取款和查询余额等功能。

假设有两位程序员，程序员乙负责定义类，编写银行账户类 Account 的类定义代码；程序员甲负责编写主函数 main，使用类 Account 来定义账户对象 obj 并访问其成员，模拟实现开户、存款、取款和查询余额等账户管理功能。在类与对象编程过程中，两位程序员分别承担了不同的角色，程序员乙定义类，程序员甲使用类定义对象。他们考虑问题的出发点是不同的，如图 7-9 所示。




程序员甲
使用类定义对象时，需了解：
1. 定义对象时，类有什么初始化方法?
2. 对象有什么公有数据成员?
3. 对象有什么公有函数成员?
4. 访问对象的公有数据成员，读写数据。
5. 调用对象的公有函数成员，处理数据。

程序员乙
为完成特定的程序功能，定义类时：
1. 需定义哪些变量，即数据成员?
2. 需定义什么函数，即函数成员?
3. 为让甲使用类，需将哪些成员开放给甲?
4. 为了让甲初始化对象，需定义什么构造函数?
5. 销毁对象如需清理内存，则定义析构函数来完成。

图 7-9 类与对象编程中的两种程序员角色

### 1. 定义类（程序员乙）

面向对象程序设计方法对程序设计任务中的客观事物进行分析、归纳，抽象出类，再对类建模。一个类模型应包含 5 个要素：

（1）描述该类事物需要哪些数据（**属性**）？

（2）对该类事物需做什么处理（**方法**）？

（3）为保证类模型被正确使用，应该如何**封装**上述属性和方法？

（4）在内存中创建对象时如何初始化（**构造方法**）？

（5）对象使用完后应释放内存，销毁对象。销毁对象时需做哪些善后处理（**析构方法**）？

C++语言使用类的语法形式来描述类模型，将属性定义成类的数据成员，将方法定义成类的函数成员。通过为类成员设定访问权限来实现类的封装。为类添加构造函数可以为该类对象提供初始化方法，添加析构函数可以在销毁对象时做某些特定的善后处理（例如释放构造函数中动态分配的内存）。

程序员乙负责定义类，编写银行账户类 Account 的类定义代码。程序员乙先设计出一个 Account 类的数据模型，属性包括账号、账户名和存款金额共 3 个数据，方法包括存款、取款和显示余额共 3 种方法。这里，我们假设程序员乙先不封装类成员，即将各成员的访问权限都设定为公有权限 public。

程序员乙用 C++语言描述银行账户类 Account，编写类定义代码。类定义代码分为类声明和类实现两部分，可以分别保存在头文件 account.h 和源程序文件 account.cpp 中。具体代码如下：

```
// 头文件：account.h
class Account                          // 类声明部分
{
public:
    int no;                            // 账号
    char name[10];                     // 账户名
    float money;                       // 存款金额
    void Deposit( );                   // 存款
    void Withdraw( );                  // 取款
    void Show( );                      // 显示余额
};

// 源程序文件：Account.cpp
```

```
#include <iostream>
using namespace std;
#include "account.h"                    // 插入头文件 account.h，声明银行账户类 Account
// 类实现部分
void Account :: Deposit( )              // 存款
{
    cout << "请输入存款金额：";
    float x;   cin >> x;
    money += x;
    Show( );                            // 显示存款后的账户余额
}
void Account :: Withdraw( )             // 取款
{
    cout << "请输入取款金额：";
    float x;   cin >> x;
    if (money < x)   cout << "账户余额不足。";
    else   money -= x;
    Show( );                            // 显示取款后的账户余额
}
void Account :: Show( )                 // 显示余额
{
    cout << "账号" << no << "的账户余额为：" << money << "元\n\n";
}
```

### 2. 使用类（程序员甲）

程序员甲负责编写主函数 main，模拟实现开户、存款、取款和显示余额等账户管理功能。主函数先输入账号、账户名和存款金额等开户信息，然后创建一个账户。创建账户的方法就是定义一个银行账户类 Account 的对象 obj。开户后，通过循环语句重复访问对象 obj 的接口来对账户进行存款、取款和查询余额的操作。程序使用如下菜单来选择操作：

1 – 存款

2 – 取款

3 – 查询余额

0 – 退出

程序员甲使用银行账户类 Account 定义对象时，应首先了解该类的接口。Account 类有哪些公有数据成员，用于保存什么数据？有哪些公有函数成员，其功能是什么？定义对象时有什么初始化方法？程序员甲需通过程序员乙编写的说明文档来了解这些问题，掌握类的使用方法。程序员乙为 Account 类定义了账号、账户名和存款金额共 3 个数据成员，还定义了存款、取款和显示余额共 3 个函数成员，并都设定为公有权限。

程序员甲用 C++语言编写主函数，函数代码保存在源程序文件 main.cpp 中。

```
// 源程序文件：main.cpp
#include <iostream>
using namespace std;
#include <string.h>                     // 插入头文件 string.h，声明系统函数 strcpy( )
#include "account.h"                    // 插入头文件 account.h，声明银行账户类 Account
int main( )                             // 主函数定义代码
```

```
{
    // 从键盘输入账号、账户名和存款金额等开户信息，先定义 3 个临时变量
    cout << "请输入开户信息（账号  账户名  存款金额）：";
    int x;    char str[10];    float y;
    cin >> x >> str >> y;

    // 创建账户 obj
    Account obj;                              // 定义一个银行账户类 Account 的对象 obj
    obj.no = x;    strcpy(obj.name, str);    obj.money = y;

    // 管理账户
    int choice;
    while ( true )
    {
        cout << "1 - 存款\n2 - 取款\n3 - 查询余额\n0 - 退出\n 请选择：";
        cin >> choice;
        if (choice == 0)    break;                    // 退出
        if (choice == 1)    obj.Deposit( );           // 存款
        else if (choice == 2)    obj.Withdraw( );     // 取款
        else if (choice == 3)    obj.Show( );         // 查询余额
    }
    return 0;
}
```

在 C++语言集成开发环境中编译连接程序员甲和乙所编写的程序代码，生成可执行程序。图 7-10 显示了该程序的执行结果。

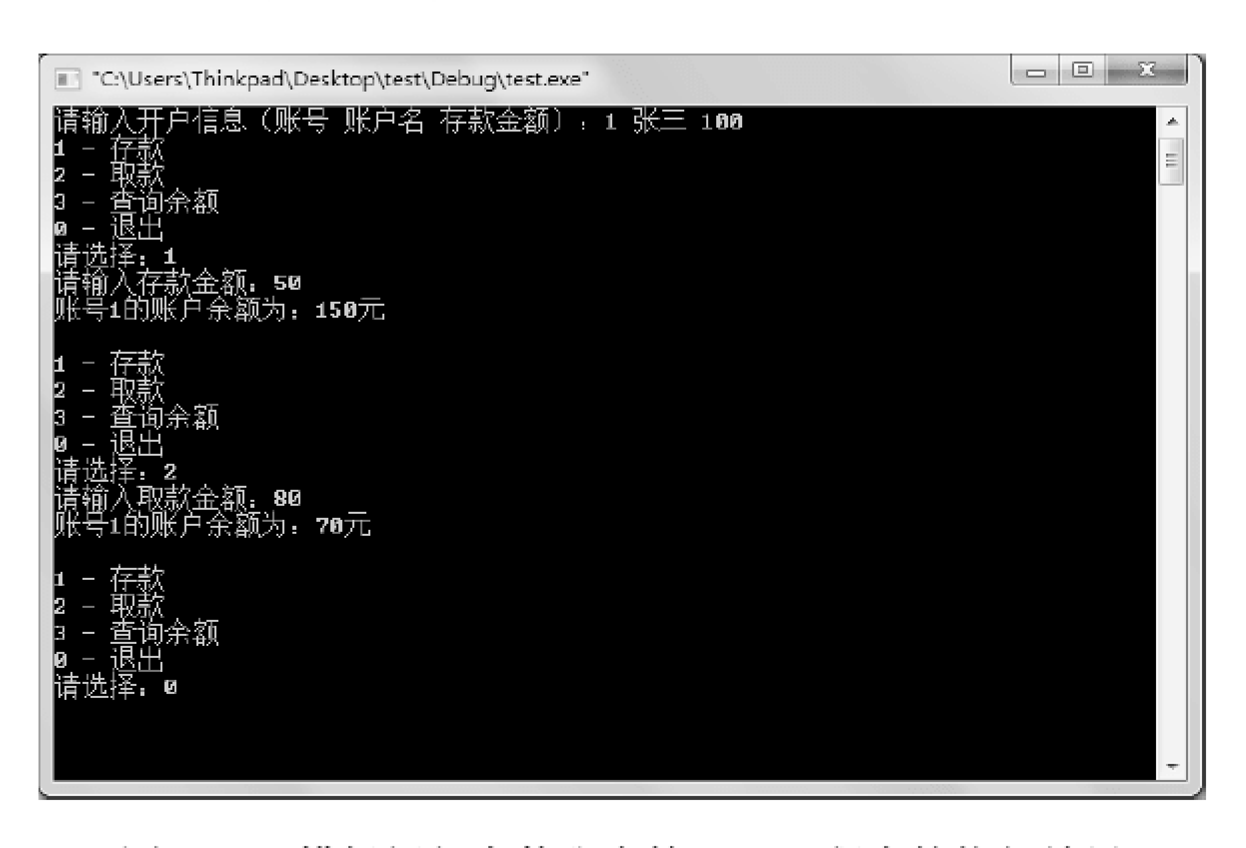



图 7-10　模拟银行存款账户管理 C++程序的执行结果

### 3. 类代码的完善与优化

程序员乙可以继续完善银行账户类 Account 的定义代码，例如添加一个构造函数：

```
Account :: Account(int iniNo, char *iniName, float iniMoney)
{
    no = iniNo;    strcpy(name, iniName);    money = iniMoney;
}
```

该构造函数为 Account 类的对象提供了初始化方法。例如程序员甲可以将创建账户 obj 的代码：

```
Account obj;                        // 定义一个银行账户类 Account 的对象 obj
obj.no = x;   strcpy(obj.name, str);   obj.money = y;
```

改写成：

```
Account obj( x, str, y );           // 定义银行账户类 Account 的对象 obj，定义时初始化
```

计算机执行该语句时将自动调用构造函数，调用时先将初始值 x、str、y 作为实参传递给形参 iniNo、iniName、iniMoney，构造函数的函数体再将这 3 个形参分别赋值给对象 obj 的数据成员 no、name 和 money。

在添加了构造函数之后，程序员甲在编写主函数时不再需要访问对象 obj 的 3 个数据成员 no、name 和 money，但仍需访问对象 obj 的 3 个函数成员 Deposit、Withdraw 和 Show。程序员乙可以优化银行账户类 Account 的封装，将 3 个数据成员隐藏起来，即将它们的访问权限设为私有权限 private；同时继续开放 3 个函数成员，即将它们的访问权限设为公有权限 public。优化后，银行账户类 Account 声明部分的代码被修改如下：

```
class Account                       // 类声明部分
{
private:                            // 隐藏 3 个数据成员
    int no;                         // 账号
    char name[10];                  // 账户名
    float money;                    // 存款金额
public:                             // 开放 3 个函数成员
    void Deposit( );                // 存款
    void Withdraw( );               // 取款
    void Show( );                   // 显示余额
    Account(int iniNo, char *iniName, float iniMoney);   // 构造函数，公有权限
};
// 类实现部分：保持不变，省略
```

编写类的程序员可以设定访问权限，开放需要访问的成员，隐藏不需要被访问的成员，这样可以避免误访问。访问权限是编写类的程序员为保证其他程序员正确访问类成员所做的封装。

## 本节习题

1. 下列关于构造函数的描述中，错误的是（　　）。
   A. 定义构造函数的目的主要是为了在创建对象时初始化对象的数据成员
   B. 构造函数在创建对象时被自动调用。每创建一个对象，构造函数即被调用一次
   C. 每个类可以定义多个构造函数，以实现不同的初始化方法
   D. 构造函数应定义为类的私有成员

2. 下列带构造函数的类 ABC，其中错误的是（　　）。

A.
```
class ABC
{
private:    int x, y;
public:
        ABC( )   {   x = 0;   y = 0;   }
        ABC(int a, int b)   {   x = a;   y = b;   }
};
```

B.
```
class ABC
{
private:    int x, y;
public:
        ABC( )   {   x = 0;   y = 0;   }
        ABC(int a, int b);
};
ABC::ABC(int a, int b)   {   x = a;   y = b;   }
```

C.
```
class ABC
{
private:    int x, y;
public:
        ABC(int a = 0, int b = 0);
};
ABC::ABC(int a, int b)   {   x = a;   y = b;   }
```

D.
```
class ABC
{
private:    int x = 0, y = 0;
public:
        void Abc(int a, int b)   {   x = a;   y = b;   }
};
```

3. 类 ABC 的默认构造函数是（　　）。

A. ABC( ) { }　　B. void ABC( ) { }

C. Abc( ) { }　　D. ABC( ) {   x = 0;   y = 0;   }

4. 已定义类 ABC：

```
class ABC
{
private:    int x, y;
public:
        ABC( )   {   x = 0;   y = 0;   }
        ABC(int a, int b)   {   x = a;   y = b;   }
};
```

执行定义对象语句“ABC obj;”会自动调用哪个构造函数？（　　）

A. ABC( )　　B. ABC(int a, int b)

C. 依次调用这两个构造函数　　D. 不调用任何构造函数

5. 已定义类 ABC：

```
class ABC
{
private:    int x, y;
public:
        ABC(int a, int b)   {   x = a;   y = b;   }
};
```

则下列定义对象语句中，错误的是（　　）。

A. ABC obj;　　B. ABC obj(5, 10);

C. ABC obj(5, 5+5);　　D. ABC Abc(0, 0);

6. 下列关于析构函数的描述中，错误的是（　　）。

A. 定义析构函数的目的是为了在销毁对象时清理对象的数据成员或其他善后工作

B. 析构函数在销毁对象时被自动调用。每销毁一个对象，析构函数即被调用一次

C. 每个类可以定义多个析构函数，以实现不同的清理方法

D. 通常，析构函数应定义为类的公有成员

7. 下列关于拷贝构造函数的描述中，错误的是（　　）。

A. 定义拷贝构造函数的目的主要是为了用一个已有的对象来初始化当前的新建对象

B. 拷贝构造函数与其他构造函数构成重载函数

C. 每创建一个对象，拷贝构造函数即被调用一次

D. 一个类如果未定义拷贝构造函数，C++将自动为该类添加一个默认拷贝构造函数

8. 已定义类 ABC：

```
class ABC
{
private:    int x, y;
public:
        ABC( )   {   x = 0;   y = 0;   }
        ABC(int a, int b)   {   x = a;   y = b;   }
        ABC(ABC &a)   {   x = a.x;   y = a.y;   }
};
```

执行定义对象语句“ABC obj1, obj2(obj1);”将会自动调用哪个构造函数？（　　）

A. ABC( )　　B. ABC(ABC &a)

C. 依次调用 ABC( )和 ABC(int a, int b)　　D. 依次调用 ABC( )和 ABC(ABC &a)

## 7.5 对象的应用

本节将分别介绍对象数组，对象的动态分配以及对象在函数中的应用。

## 7.5.1 对象数组

可以定义一个对象数组来保存多个对象。例 7-8 给出一个正方形类 Square 对象数组的 C++演示程序。

**例 7-8 一个正方形类 Square 对象数组的 C++演示程序**

```
1   #include <iostream>
2   using namespace std;
3
4   class Square                        // 定义一个正方形类 Square
5   {
6   public:
7       double a;                       // 保存边长的数据成员 a
8       double Area( )                  // 求正方形面积的函数成员 Area，内联函数
9       {    return ( a*a );    }
10      Square( double x = 0 )          // 带默认形参值的构造函数，内联函数
11      {    a = x;    }
12  };
13
14  int main( )
15  {
16      Square   obj[3] = {   Square(2), Square(3), Square（4） };
17      // 这条语句定义了一个正方形类 Square 的对象数组 obj，定义时初始化
18      // 数组 obj 有 3 个元素，每个元素都是一个正方形对象，边长分别初始化为 2、3、4
19
20      for (int n = 0; n < 3; n++)   // 显示数组中各正方形对象的边长和面积
21      {
22          cout << obj[n].a << endl;
23          cout << obj[n].Area( ) << endl;
24      }
25      return 0;
26  }
```

### 1. 对象数组的定义

定义对象数组与定义普通数组的语法形式基本相同。例如，定义一个正方形类 Square 的一维对象数组 obj：

```
Square   obj[3];                    // 对象数组 obj 包含 3 个元素，每个元素都是一个正方形对象
```

定义对象数组时可以初始化数组元素。例如，例 7-8 的代码第 16 行：

```
Square   obj[3] = {   Square(2), Square(3), Square（4）  };
```

对象初始值的表示形式是“**类名( 实参列表 )**”。例如，Square(2)、Square(3)和 Square(4)分别表示边长为 2、3、4 的正方形对象。

计算机执行对象数组定义语句时，将逐个构造数组中的对象元素。数组元素是属于同

一个类的对象，每构造一个数组元素都会自动调用一次该类的构造函数。例如，下列定义对象数组语句：

```
Square   obj[3] = {   Square(2), Square（3）  };      // 部分初始化，只初始化了前 2 个元素
```

这条语句所定义的对象数组 obj 包含 3 个元素，但定义时只给了 2 个初始值，即部分初始化。计算机执行该语句时仍然会调用 3 次类 Square 的构造函数，其中前 2 次调用分别传递了实参，将 obj[0]、obj[1]的边长初始化为 2、3；第 3 次调用时未传递实参，计算机会自动使用默认形参值将 obj[2]的边长初始化为 0。

计算机执行对象数组定义语句时，数组中有多少个元素，就会调用多少次构造函数。

#### 2. 对象数组的访问

对象数组中的每个元素都是一个对象，每个对象都有各自的成员。可以访问各对象元素的公有成员，其访问形式是：

**对象数组名[下标]．公有成员名**

例如，例 7-8 的代码第 22 行和第 23 行。

```
cout << obj[n].a << endl;              // 访问第 n 个元素的边长 a
cout << obj[n].Area( ) << endl;        // 调用第 n 个元素的求面积函数 Area
```

也可以使用对象指针访问对象数组中的元素。例如，例 7-8 的代码 20~24 行可改写为：

```
Square *p = obj;               // 定义一个类 Square 的对象指针，初始化指向对象数组 obj 的首地址
for (int n = 0; n < 3; n++)
{
      cout << p->a << endl;           // 通过指针 p 间接访问当前所指向元素的边长 a
      cout << p->Area( ) << endl;     // 通过指针 p 间接调用当前所指向元素的函数 Area
      p++;                            // 将对象指针 p 加 1，指向数组中的下一个对象元素
}
```

#### 3. 对象数组的析构

执行例 7-8 的程序，当程序执行结束即将退出时，计算机将销毁主函数中定义的对象数组 obj，释放其内存空间。

销毁对象数组，就是逐个销毁其中的所有对象元素。对象数组中的元素是属于同一个类的对象，每销毁一个对象都会自动调用一次该类的析构函数。换句话说，计算机销毁对象数组时，数组中有多少个元素，就会调用多少次析构函数。

### 7.5.2　对象的动态分配

可以使用动态内存分配方法定义对象或对象数组。C++语言使用 new 运算符动态分配内存，内存使用完之后应使用 delete 运算符及时释放，提高内存利用率。内存的动态分配、访问及释放都需要通过指针变量才能实现。

#### 1. 单个对象的分配与释放

使用 new 运算符动态分配单个对象的内存时需要指定类类型，分配成功后将返回所分配内存单元的首地址。需要预先定义一个同类的对象指针来保存这个首地址，后续访问内存单元时将使用该对象指针进行间接访问，释放内存单元时也要使用对象指针来指定释放哪个内存单元。例如：

```
Square *p;              // 为了动态分配一个类 Square 对象，需预先定义一个同类的对象指针 p
p = new Square;         // 动态分配一个类 Square 的对象，用 p 保存所分配内存的首地址
…                       // 通过对象指针间接访问对象的公有成员，“p->成员名”或“(*p).成员名”
delete p;               // 删除对象，释放其内存空间
```

动态分配单个对象时可以初始化，例如：

```
p = new Square( 2 );    // 动态分配一个类 Square 的对象，并将其边长初始化为 2
```

动态分配某个类的对象时会自动调用该类的构造函数，删除对象时会自动调用该类的析构函数。

#### 2. 对象数组的分配与释放

使用 new 运算符动态分配对象数组，可以在程序执行时根据需要分配适量的内存，减少内存占用。内存使用完之后应及时释放，提高内存利用率。例如：

```
Square *p;              // 为了动态分配一个类 Square 的对象数组，需预先定义对象指针 p
p = new Square[3];      // 动态分配一个类 Square 的一维对象数组，包含 3 个元素
p[0].a = 2;    cout << p[0].Area( );          // 可通过下标运算符访问对象元素的成员
(p+1)->a = 3;    cout << (p+1)->Area( );      // 或通过指针运算符访问对象元素的成员
…
delete [ ]p;            // 删除对象数组，释放其内存空间
```

动态分配对象数组时，数组中有多少个元素就会调用多少次构造函数。删除对象数组时，数组中有多少个元素就会调用多少次析构函数。

### 7.5.3 对象作为函数的形参

定义在函数内部的对象是局部对象，其他函数不能直接访问。对象可以作为参数在函数间传递。调用函数时，主调函数和被调函数之间需要通过形实结合来传递对象，将保存在主调函数里的对象以**对象实参**的形式传递给被调函数的**对象形参**。例 7-9 给出一个求正方形对象内切圆面积的 C++程序。

**例 7-9 一个求正方形对象内切圆面积的 C++程序**

```
#include <iostream>
using namespace std;

class Square                    // 定义一个正方形类 Square
{
public:
```

```
7       double a;                    // 保存边长的数据成员 a
8       double Area( )               // 求正方形面积的函数成员 Area，内联函数
9       {   return ( a*a );   }
10      Square( double x = 0 )       // 带默认形参值的构造函数，内联函数
11      {   a = x;   }
12  };
13
14  double InnerCircleArea( Square s )   // 求正方形对象 s 的内切圆面积
15  {
16      double r = s.a / 2;          // 内切圆直径等于正方形边长，因此半径 r = s.a / 2
17      return (3.14*r*r);           // 返回内切圆的面积
18  }
19
20  int main( )
21  {
22      Square   obj(10);            // 定义一个正方形对象 obj，边长初始化为 10
23      cout << InnerCircleArea( obj ) << endl;   // 调用函数求对象 obj 的内切圆面积
24      return 0;
25  }
```

例 7-9 通过形实结合将主函数中定义的正方形对象 obj 传递给函数 InnerCircleArea 中的对象形参 s，这个形实结合过程相当于执行了如下语句：

```
Square s( obj );
```

即定义一个正方形对象 s，为 s 分配内存空间为并用 obj 进行初始化。这个初始化是通过自动调用类 Square 的默认拷贝构造函数完成的。

5.5 节曾介绍过函数间参数传递的三种方式，分别是值传递、引用传递和指针传递。下面将分别介绍在函数间传递对象时的值传递、引用传递和指针传递。

### 1. 值传递与常对象

例 7-9 传递对象所使用的就是值传递方式。对象形参 s 单独分配内存，另外保存一份对象实参 obj 的数据副本，被调函数访问的是对象形参 s 中的副本。

从功能上讲，函数 InnerCircleArea 只需要读取对象形参 s 中的数据，而不需要做任何修改的操作。像常变量一样，可以将任何只读不改的对象定义成**常对象**。常对象定义时必须初始化，定义后不能再修改其数据成员。如果修改常对象中的数据成员，编译器在编译时将提示错误信息。例如：

```
const Square   obj(2);          // 定义常对象 obj，边长初始化为 2
cout << obj.a << endl;          // 正确：读取常对象 obj 中的数据成员 a
obj.a = 5;                      // 错误：不能修改常对象 obj 中的数据成员
```

可将例 7-9 中函数 InnerCircleArea 的函数头（代码第 14 行）改成使用如下常对象形参：

```
double InnerCircleArea( const Square s )     // 将形参 s 定义成常对象
```

值传递实际上是重新构造了一个对象形参，这需要花费执行时间和内存空间。相比较而言，引用传递和指针传递具有更高的数据传递效率。

### 2. 引用传递与常引用

引用传递将被调函数的对象形参定义成主调函数中对象实参的引用，被调函数通过该引用间接访问主调函数中的对象。例如，可将例 7-9 中的函数 InnerCircleArea 改成如下的引用传递方式：

```
double InnerCircleArea( Square &s )          // 引用传递：通过 s 引用主函数中的对象 obj
{
    double r = s.a / 2;                      // 内切圆直径等于正方形边长，因此半径 r = s.a / 2
    return (3.14*r*r);                       // 返回内切圆的面积
}
```

改为引用传递后，主函数中的调用语句保持不变，仍为：

```
cout << InnerCircleArea( obj ) << endl;      // 调用时的实参为对象 obj
```

引用传递可以提高数据传递效率，但也有副作用。例如，函数 InnerCircleArea 中任何对引用 s 的修改都会影响到所引用的主函数对象 obj。如果希望避免这种因引用传递所带来的函数间相互影响，可以将上述函数头中的引用定义成**常引用**：

```
double InnerCircleArea( const Square &s )  // 引用传递：将 s 定义成常引用
```

定义成常引用之后，任何通过 s 间接修改所引用对象 obj 中数据成员的操作都是错误的，编译器在编译时将会提示该错误。

引用传递是函数间传递对象参数时最常用的形式。

### 3. 指针传递与指向常对象的指针

指针传递将主调函数中对象实参的内存地址传给被调函数的对象指针形参，被调函数通过该地址间接访问主调函数中的对象。例如，可将例 7-9 中的函数 InnerCircleArea 改成如下的指针传递方式：

```
double InnerCircleArea( Square *s )          // 指针传递：通过指针 s 间接访问主函数中的对象 obj
{
    double r = s->a / 2;                     // 内切圆直径等于正方形边长，因此半径 r = s->a / 2
    return (3.14*r*r);                       // 返回内切圆的面积
}
```

改为指针传递后，主函数中的调用语句相应地改为：

```
cout << InnerCircleArea( &obj ) << endl;     // 调用时的实参为对象 obj 的内存地址
```

和引用传递一样，指针传递可以提高数据传递效率，也有副作用。如果希望避免因指针传递所带来的函数间的相互影响，可以将上述函数头中的对象指针形参定义成**指向常对象的指针**：

```
double InnerCircleArea( const Square *s )          // 指针传递：将 s 定义成指向常对象的指针
```

定义成指向常对象的指针之后，任何通过 s 间接修改所指向对象 obj 中数据成员的操

作都是错误的，编译器在编译时将会提示该错误。

函数间传递对象有三种方式，分别是值传递、引用传递和指针传递。如果功能设计上被调函数不需要修改主调函数传递过来的对象，而只是读取其中的数据，那么程序员可以主动地将被调函数的对象形参定义成常对象、常引用或指向常对象的指针。这时，如果被调函数的函数体中含有修改对象的语句，编译时编译器将会提示错误信息，从而帮助程序员迅速排查出这类错误。

## 本节习题

1. 已定义一个圆形类 Circle：

```
class Circle
{
private:    double r;
public:
        void SetR(double x)    {    r = x;    }
        double GetArea( )    {    return 3.14*r*r;    }
};
```

用 Circle 类定义一个对象数组“Circle c[3];”，则下列语句中错误的是（　　）。

A. for (int i = 0; i < 3; i++)
```
   {    c[i].SetR(i*2.5);    cout << c[i].GetArea( ) << endl;    }
```
B. for (int i = 3; i >= 0; i--)
```
   {    c[i].SetR(i*2.5);    cout << c[i].GetArea( ) << endl;    }
```
C. Circle *p = c;
```
    for (int i = 0; i < 3; i++)
   {    p->SetR(i*2.5);    cout << p->GetArea( ) << endl;    p++;    }
```
D. Circle *p = &c[2];
```
    for (int i = 2; i >= 0; i--)
   {    p->SetR(i*2.5);    cout << p->GetArea( ) << endl;    p--;    }
```

2. 使用类 ABC 做定义“ABC x, *p, y[3];”，执行该定义语句将自动调用几次类 ABC 的构造函数？（　　）

A. 0　　B. 3　　C. 4　　D. 5

3. 已定义一个圆形类 Circle：

```
class Circle
{
private:    double r;
public:
        void SetR(double x)    {    r = x;    }
        double GetArea( )    { return 3.14*r*r;    }
};
```

下列语句中正确的是（　　）。

A. Circle c, *p = &c;　p.SetR(10.5);　p.GetArea( );

B. Circle *p;　p->SetR(10.5);　p->GetArea( );

C. Circle *p;　p = new Circle;　p->SetR(10.5);　p->GetArea( );　delete p;

D. Circle *p = new Circle;　p->SetR(10.5);　p->GetArea( );　delete [ ]p;

4. 函数间传递对象数据不能采用下列哪种方式？（　　）

A. 值传递　　B. 引用传递

C. 指针传递　　D. 被调函数直接访问主调函数中的局部对象

5. 通过值传递在函数间传递对象数据，形实结合时会自动调用下列哪个构造函数来初始化对象形参？（　　）

A. 不带形参的构造函数　　B. 带形参的构造函数

C. 拷贝构造函数　　D. 析构函数

## 7.6 类中的常成员与静态成员

常变量、常对象、常引用、指向常变量或常对象的指针等在定义时都使用了 const 关键字，这是 C++语言引入的一种数据保护机制，称为 const **数据保护机制**。例如，如果功能设计上被调函数只是使用主调函数传递过来的数据，而不需要修改，那么程序员在编写程序时可以使用 const 关键字主动地对被调函数形参进行限定，限定被调函数不能修改主调函数传递过来的数据。在定义类的时候，也可以使用 const 数据保护机制来保护数据成员不被修改，本节将介绍类中的常成员。

C++语言中的静态（static）是与作用域相关的一个概念。例如，静态可以将全局变量的作用域限定在本程序文件范围内访问，也可以将全局变量的作用域限定在某个函数范围内访问（此时全局变量就变成了该函数内部的一个静态局部变量），我们将其统称为 static **静态机制**。

本节通过一个模拟出租车管理的程序实例来具体讲解类中常成员、静态成员的应用场景和语法细则。某出租车公司有 10 辆出租车，假设现在有 100 个打车订单需要处理，以随机方式选派执行订单的车辆。要求采用面向对象程序设计方法编写一个模拟出租车管理的 C++程序，计算 100 个打车订单处理完之后每辆出租车的收费总额，以及整个公司的总收费金额。例 7-10 给出了完整的模拟出租车管理的 C++演示程序代码。

**例 7-10　一个模拟出租车管理的 C++演示程序**

```
#include <iostream>
using namespace std;
#include <stdlib.h>                  // 插入随机数生成函数 rand 的声明头文件 stdlib.h
int GetARand(int a, int b)           // 外部函数：在某个特定区间[a, b]内生成随机数
{   return ( rand( ) %(b-a+1) +a );   }

void AddTotal( int f );   // 类 Taxi 要调用这个函数，但它定义在后面，需先声明其原型
class Taxi                           // 定义一个出租车类 Taxi
```

```
9    {
10   private:
11       int   price;                   // 数据成员：保存里程单价
12       int   fare;                    // 数据成员：保存收费总额
13   public:
14       void SetPrice(int p)   {   price = p;   }         // 函数成员：设置里程单价
15       int GetPrice( )   {   return price;   }            // 函数成员：读取里程单价
16       void SetFare(int f)   {   fare = f;   }            // 函数成员：设置收费总额
17       int GetFare( )   {   return fare;   }              // 函数成员：读取收费总额
18       void AddFare(int f)   {   fare += f;   }           // 函数成员：累加收费总额
19       void Order( )   // 函数成员：执行一个打车订单
20       {
21           int x = GetARand(5, 100);         // 调用外部函数 GetARand 随机生成一个公里数
22           AddFare( x*price );               // 调用函数成员 AddFare 累加本车收费总额
23           AddTotal( x*price );              // 调用外部函数 AddTotal 累加公司总收费金额
24       }
25       Taxi(int p=0, int f=0)                // 构造函数：带默认形参值 0
26       {   price = p;   fare = f;   }
27   };
28
29   int   totalFare = 0;                      // 全局变量：保存出租车公司总的收费金额
30   void AddTotal( int f )   {   totalFare += f;   }      // 外部函数：累加公司总收费金额
31
32   int main( )
33   {
34       int n, x;
35       Taxi tObj[10];        // 定义一个 Taxi 类的对象数组，可保存 10 辆出租车的相关数据
36       for (n = 0; n < 10; n++)              // 设置出租车的初始值
37       {
38           x = GetARand(2, 5);   tObj[n].SetPrice( x );       // 随机生成每辆车的里程单价
39           tObj[n].SetFare( 0 );             // 每辆车的初始收费总额都为 0
40       }
41       for (n = 0; n < 100; n++)             // 模拟处理 100 个打车订单
42       {
43           x = GetARand(0, 9);               // 随机选择一个 0~9 的车号，赋值给 x
44           tObj[x].Order( );                 // 选派第 x 辆车执行订单
45       }
46       for (n = 0; n < 10; n++)              // 显示每辆车的里程单价和收费总额
47           cout << tObj[n].GetPrice( ) << ", " << tObj[n].GetFare( ) << endl;
48       cout << totalFare << endl;            // 显示公司总收费金额
49       return 0;
50   }
```

例 7-10 程序的代码说明如下。

（1）**出租车类** Taxi（**代码第** 8~27 **行**）。根据程序功能要求，出租车类 Taxi 提炼出 2 个数据成员，即里程单价 price 和收费总额 fare。出租车类 Taxi 还定义了 2 个函数成员，分别是执行订单的 Order 和累加收费总额的 AddFare，完成订单处理功能。在执行打车订单时，函数 Order 首先随机生成一个公里数，然后计算车费，并分别累加到本车收费总额和公司总收费金额中。另外，类 Taxi 还定义了一个带默认形参值的构造函数。

（2）**封装出租车类** Taxi。出租车类 Taxi 将数据成员里程单价 price 和收费总额 fare 设为 private 权限（即隐藏起来），然后为这两个被隐藏的数据成员提供设置和读取方法，它们分别是 SetPrice 和 GetPrice、SetFare 和 GetFare。因为这 4 个函数成员都比较简短，为了缩小程序篇幅，将它们的函数头和函数体写在了一行。这种书写形式不会影响函数语法和功能的正确性。

（3）**生成随机数**。C++语言提供一个生成随机数的系统函数 rand，使用这个函数时需插入其声明头文件 stdlib.h。例 7-10 在函数 rand 的基础上又定义了一个外部函数 GetARand（代码第 4~5 行），它可以在某个特定区间[a, b]内生成随机数。程序将使用这个外部函数来随机生成打车订单千米数（5~100 千米）、每辆车的里程单价（2~5 元），以及选派执行订单车辆的车号（0~9）。

（4）**出租车公司的总收费金额**。例 7-10 定义了一个全局变量 totalFare（代码第 29 行）来保存出租车公司总的收费金额，另外还定义了一个外部函数 AddTotal（代码第 30 行）来累加公司总收费金额。

（5）**主函数**。主函数定义了一个 Taxi 类的对象数组 tObj（代码第 35 行），包含 10 个数组元素，然后用它来保存 10 辆出租车的数据。随机设置每辆出租车的里程单价，并将初始收费总额都设为 0（代码第 36~40 行）。使用循环结构模拟处理 100 个打车订单（代码第 41~45 行）。每次处理订单都是以随机抽选方式安排车辆，然后执行订单。程序最后显示每辆出租车的里程单价、收费金额以及公司总的收费金额。

图 7-11 给出一个执行例 7-10 程序的示意结果。

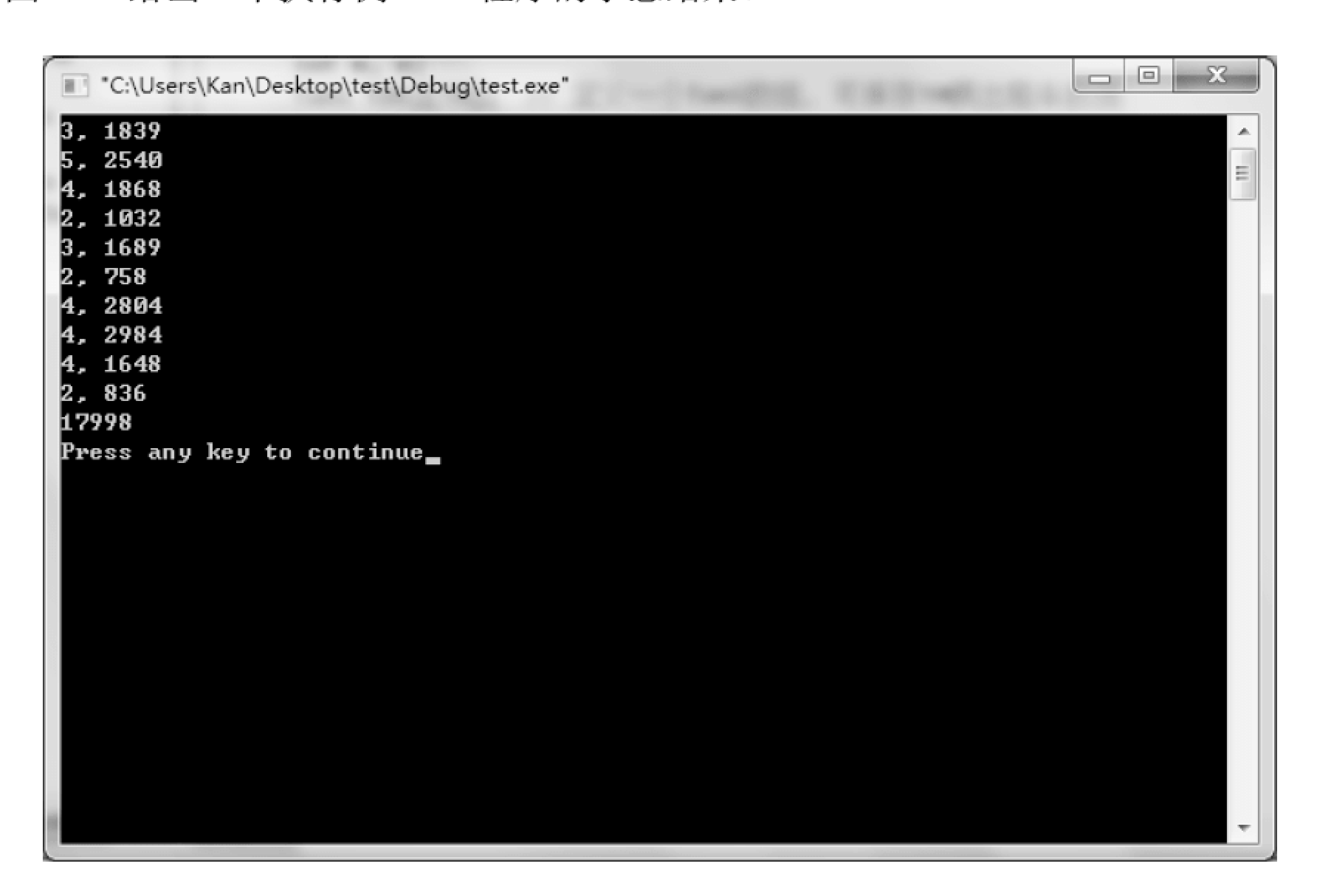



图 7-11 执行例 7-10 程序的示意结果

在理解了例 7-10 的模拟出租车管理程序之后，下面就通过这个实例来具体讲解类中常成员、静态成员的应用场景和语法细则。

## 7.6.1 常成员

定义类时，使用 const 关键字进行限定的成员称为**常成员**。数据成员和函数成员都可以定义成常成员。在类中定义常成员的目的是保护该类对象中的数据成员在程序执行过程中不被修改。

### 1. 常数据成员

分析例 7-10 中出租车类 Taxi 的数据成员 price，其中保存的是出租车里程单价。通常，每辆出租车的里程单价都只是在初始化时设置一次，在程序后面的订单处理流程中不会再改变。

如果一个数据成员所保存的数值在初始化以后不会改变，那么可以将这个数据成员定义成常数据成员。换句话说，如果定义成常数据成员，那么该成员只能在初始化时赋值，初始化后不能再次赋值。常数据成员的含义与常变量大致相同，所不同的是在类中声明常数据成员时不能直接初始化。类中的任何数据成员都不能在声明时直接赋初始值，必须通过构造函数完成初始化。

如果将例 7-10 中出租车类 Taxi 的里程单价 price 定义成常数据成员，那就要对程序代码做几处修改。这些修改都是 C++语言针对常数据成员所提出的语法要求。例 7-11 给出了修改后的示意代码，其中与常数据成员语法无关的部分被省略掉了。

**例 7-11 一个模拟出租车管理的 C++演示程序（常数据成员 price）**

```
//此处代码省略
class Taxi                    // 定义一个出租车类 Taxi
{
private:
    const int   price;        // 修改 1：声明 price 时加 const 关键字，常数据成员
    int   fare;               // 数据成员：保存收费总额
public:
    void SetPrice(int p)   {   price = p;   }   // 修改 2：常数据成员不能赋值，删除线表示删除
    int GetPrice( )   {   return price;   }     // 函数成员：读取里程单价
    //此处代码省略
    Taxi(int p=0, int f=0) : price(p)           // 修改 3：在初始化列表中初始化常数据成员 price
    {   price = p;   fare = f;   }              // 在函数体中初始化其他数据成员
};
//此处代码省略

int main( )
{
    int n, x;
    // 修改 4：定义 Taxi 类对象数组时必须初始化，重点是给出里程单价的初始值
    Taxi tObj[10] = {   Taxi(GetARand(2, 5), 0), Taxi(GetARand(2, 5), 0),
                        Taxi(GetARand(2, 5), 0), Taxi(GetARand(2, 5), 0), Taxi(GetARand(2, 5), 0),
                        Taxi(GetARand(2, 5), 0), Taxi(GetARand(2, 5), 0), Taxi(GetARand(2, 5), 0),
                        Taxi(GetARand(2, 5), 0), Taxi(GetARand(2, 5), 0)   };
```

```
40                     // 随机生成每辆车的里程单价，并将初始收费总额都设为 0
       Taxi tObj[10];   // 修改 5：删除原来的数组定义和设置初始值代码
       for (n = 0; n < 10; n++)   // 设置出租车的初始值
       {
           x = GetARand(2, 5);   tObj[n].SetPrice( x );   // 随机生成每辆车的里程单价
           tObj[n].SetFare( 0 );   // 每辆车的初始收费总额都为 0
41~49  }
50     //此处代码省略
    }
```

常数据成员的语法细则：

（1）**关键字** const。在类中声明常数据成员时需使用关键字 const 进行限定，声明时不能初始化。例如，例 7-11 中的代码第 11 行。

（2）**初始化列表**。类中的任何函数都不能对常数据成员赋值，包括构造函数。为构造函数添加初始化列表是对常数据成员进行初始化的唯一途径。在构造函数的函数头后面添加初始化列表，其语法形式如下：

```
构造函数名( 形参列表 ) : 常数据成员名 1( 形参 1 ), 常数据成员名 2( 形参 2 ), …
{
… // 在函数体中初始化其他数据成员
}
```

其中，形参 1、形参 2 等是从形参列表中提取出来的，并在初始化列表中进行二次接力传递。例如例 7-11 中的代码第 25 行，在构造函数头后面添加初始化列表“: price(p)”是唯一能够设置常数据成员 price 初始值的地方，任何其他地方都不能对 price 再次赋值（例如例 7-11 中的代码第 14 行和第 26 行，删除线表示删除）。

（3）**定义对象时初始化**。定义含常数据成员类的对象时需要初始化，重点是给出常数据成员的初始值。对象初始化后，其中的常数据成员只能读取，不能修改。例如，例 7-11 中的代码第 36~39 行，在定义 Taxi 类对象数组 tObj 时通过初始化设置了每辆出租车的里程单价和收费总额。其中的里程单价就是常数据成员，初始化后不能修改。

程序员在设计类的时候，如果认为某个数据成员所保存的数值在初始化以后不会改变，那么可以主动将这个数据成员定义成常数据成员。定义成常数据成员后，原有的程序功能完全不受影响。只要定义成常数据成员，编译时编译器会帮助程序员排查出任何修改（或潜在的可能修改）该数据成员的语句，从而减轻程序员排查此类错误的工作量。

### 2. 常函数成员

如果某个函数成员只需要读取类中数据成员的数值，即不会修改它们，那么可以将该函数定义成**常函数成员**。换句话说，如果将某个函数成员定义成常函数成员，那么该函数就只能读取类中数据成员的数值，不能修改它们（例如赋值或 cin 输入）。

例 7-10 中出租车类 Taxi 的函数成员 GetPrice 和 GetFare 都满足常函数成员的条件，可以将它们定义成常函数成员。例如，可以按照如下两种形式将 GetPrice 定义成常函数成员。

（1）**内联函数**。在类声明部分直接定义的函数将被当作内联函数处理，此时在函数头

后面加 const 关键字就可以将它定义成常函数成员。例如，修改例 7-10 中代码第 15 行 GetPrice 函数的定义代码：

```
int GetPrice( ) const   {   return price;   }     // 常函数成员：读取里程单价
```

（2）**非内联函数**。如果在类声明部分声明函数原型，在类实现部分给出完整的函数定义代码，此时定义常函数成员需要在其声明和定义语句的函数头后面分别加上 const 关键字。例如，将常函数成员 GetPrice 的声明和实现分开：

```
int GetPrice( ) const ;                    // 在类的声明部分声明常函数成员
int Taxi :: GetPrice( ) const              // 在类的实现部分定义常函数成员
{   return price;   }
```

常函数成员的语法细则：

（1）声明、定义常函数成员须在函数头后面加关键字 const 进行限定。

（2）常函数成员只能读取类中数据成员的数值，不能修改它们（例如赋值或 cin 输入）。

（3）常函数成员只能调用其他常函数成员。换句话说，常函数成员不能调用其他无 const 限定的函数成员，以防止这些函数间接修改了数据成员。

（4）通过常对象只能调用其常函数成员。换句话说，通过常对象不能调用无 const 限定的函数成员，以防止这些函数间接修改了常对象的数据成员。

（5）除形参的个数和类型之外，还可以用关键字 const 区分类中的重载函数。

## 7.6.2　静态成员

定义类时，使用关键字 static 进行限定的成员称为**静态成员**。数据成员和函数成员都可以定义成静态成员。C++语言中的 static 静态机制是与作用域相关的一个概念。例如，静态可以将全局变量的作用域限定在本程序文件范围内访问，也可以将全局变量的作用域限定在某个函数范围内访问（此时全局变量就变成了该函数内部的一个静态局部变量）。那么类中的静态数据成员或静态函数成员又是什么含义，其作用又是什么呢？

面向对象程序设计希望用类管理所有的程序代码，程序中没有游离在类外的全局变量或外部函数。对于一些共用的全局变量或外部函数，可以将它们划归到某个具有关联关系的类中，与类中的其他成员一起进行统一管理。但这些全局变量或外部函数与类中的其他成员是有区别的，C++语言使用 static 关键字将它们定义成静态成员。

### 1. 静态数据成员

例 7-10 中代码第 29 行定义了一个全局变量 totalFare 来保存出租车公司总的收费金额。可以把这个全局变量划归成出租车类 Taxi 的下属数据成员，与其中的里程单价 price 和收费总额 fare 一起进行统一管理。这时应当使用 static 关键字将全局变量 totalFare 声明成出租车类 Taxi 的静态数据成员，同时还要对其他几处程序代码做相应修改。这些修改都是 C++语言针对静态数据成员所提出的语法要求。例 7-12 给出了修改后的示意代码，其中与静态数据成员语法无关的部分被省略掉了。

**例 7-12　一个模拟出租车管理的 C++演示程序（静态数据成员 totalFare）**

```
1~7   //此处代码省略
8     class Taxi                          // 定义一个出租车类 Taxi
9     {
10    private:
11          int   price;                  // 数据成员：保存里程单价
12          int   fare;                   // 数据成员：保存收费总额
13    public:
14~24       //此处代码省略
25          Taxi(int p=0, int f=0)        // 构造函数：带默认形参值 0
26          {   price = p;   fare = f;   }
27          static int   totalFare;   // 修改 1：将全局变量 totalFare 声明成类 Taxi 的静态数据成员
28    };
29
30    int   Taxi :: totalFare = 0;   // 修改 2：在类实现部分定义并初始化静态数据成员 totalFare
31    void AddTotal( int f )   {   Taxi :: totalFare += f;   }   // 修改 3：在外部函数中访问 totalFare
32
33    int main( )
34    {
35~48       //此处代码省略
49          cout << Taxi :: totalFare << endl;   // 修改 4：主函数访问 totalFare 显示公司总收费金额
50          return 0;
51    }
```

静态数据成员的语法细则：

（1）**关键字** static。在类中声明静态数据成员需使用关键字 static 进行限定，声明时不能初始化。例如，例 7-12 中的代码第 27 行。

（2）**定义及初始化**。必须在类声明的大括号后面（通常是和函数成员的定义代码一起放在类实现部分）对静态数据成员进行定义，定义时不能再加关键字 static。定义时可以初始化。例如，例 7-12 中的代码第 30 行。

（3）**在类的函数成员中访问**。类中的函数成员直接使用成员名访问静态数据成员，访问时不受权限约束。这一点与访问普通数据成员是一样的。

（4）**在类外其他函数中访问**。在类外其他函数（例如主函数）中访问静态数据成员需以“类名 :: 静态数据成员名”的形式访问，或通过任何一个该类对象以“对象名.静态数据成员名”的形式访问，或通过任何一个该类对象指针以“对象指针名->静态数据成员名”的形式访问。例如，

```
cout << Taxi :: totalFare << endl;     // 通过类名 Taxi 直接访问静态数据成员 totalFare
```

或先定义对象（或对象指针），再通过对象名（或对象指针）访问，访问形式如下：

```
Taxi   obj, *p = &obj;                 // 定义一个 Taxi 类的对象 obj 和对象指针 p
cout << obj. totalFare << endl;        // 通过对象 obj 访问静态数据成员 totalFare
cout << p-> totalFare << endl;         // 通过对象指针 p 访问静态数据成员 totalFare
```

类外访问受权限约束，只能访问公有的静态数据成员。访问权限决定了静态数据成员的作用域。私有静态数据成员具有类作用域，只能在类内访问。公有静态数据成员具有文

件作用域，可以被本文件中的任何函数访问，并且可通过类声明将其作用域扩展到任何程序文件。

（5）**内存分配**。和全局变量一样，静态数据成员也是静态分配的，被分配在静态存储区。静态数据成员在程序加载后立即分配内存，直到程序执行结束退出时才被释放。这一点与类中其他普通数据成员是完全不同的。例如，例 7-12 在主函数中定义了一个出租车类 Taxi 的对象数组 tObj：

```
Taxi   tObj[10];                    // 定义一个 Taxi 类的对象数组，包含 10 个对象元素
```

计算机执行这条语句时将在内存中创建 10 个对象元素，即为它们分配内存空间。每个对象元素所占用的字节数等于 Taxi 类中所有数据成员占用字节数的总和，但是**请注意**：这些数据成员不包括静态数据成员。图 7-12 给出了对象数组 tObj 的内存分配示意图。

| | | 各数据成员在出租车类Taxi中的定义 |
|---|---|---|
| 静态存储区：Taxi::totalFare | 4 字节 | static int totalFare; |
| 以下为自动存储区 | …… | |
| tObj[0] | 4 字节 | int price; |
| | 4 字节 | int fare; |
| …… | 4 字节 | int price; |
| | 4 字节 | int fare; |
| tObj[9] | 4 字节 | int price ; |
| | 4 字节 | Int fare; |

图 7-12　出租车对象数组 tObj 的内存分配示意图

从图 7-12 中可以看出，计算机为对象分配内存时只会为普通数据成员分配内存。出租车对象 tObj[0]~tObj[9]都包含里程单价 price 和收费总额 fare 这两个普通数据成员，但并不包含静态数据成员 totalFare。出租车类的静态数据成员 Taxi::totalFare 是在静态存储区单独分配内存的。

一方面，静态数据成员不依托于某个具体的内存对象，程序加载时就在静态存储区单独分配内存，直到程序执行结束退出时才被释放；另一方面，无论用类定义多少个对象，静态数据成员只会在内存中分配一个内存单元。换句话说，用类定义多个对象，每个对象的普通数据成员都各自分配内存单元，但所有对象的静态数据成员会共用同一个内存单元。

例如，数组元素 tObj[0].price~tObj[9].price 和 tObj[0].fare~tObj[9].fare 都有各自的内存单元，但 tObj[0].totalFare~tObj[9].totalFare 会共用同一个内存单元，这就是静态存储区中的 Taxi:: totalFare，也就是说，访问 tObj[0]~tObj[9]中任一对象元素的静态数据成员 totalFare 访问的都是 Taxi:: totalFare。

### 2. 静态函数成员

可以将全局变量划归到某个具有关联关系的类中，作为类的静态数据成员进行管理。同样，也可以将外部函数划归到某个具有关联关系的类中，作为类的静态函数成员进行

管理。

例如，例 7-12 在将全局变量 totalFare 划归成出租车类 Taxi 的静态数据成员之后，可以进一步把与之关联的外部函数 AddTotal 也划归到出租车类 Taxi 中，作为类的静态函数成员来进行管理。将外部函数 AddTotal 定义成出租车类 Taxi 的下属静态函数成员，需对例 7-12 中的两处代码做修改。

（1）在出租车类 Taxi 的声明部分添加如下的原型声明语句：

```
static    void AddTotal( int f );          // 声明时加 static 关键字
```

在例 7-12 中，只有出租车类 Taxi 的函数成员 Order 会调用函数 AddTotal，即 AddTotal 只在类 Taxi 的内部使用，因此可以将其访问权限设为 private。

（2）在出租车类 Taxi 的实现部分给出函数 AddTotal 的完整定义代码。在例 7-12 中，就是将代码第 31 行改成如下的形式：

void **Taxi** :: AddTotal( int f )   {   ~~Taxi ::~~ totalFare += f;   }   // 此处不能再加 static 关键字

在函数名前面加“Taxi ::”，表示函数 AddTotal 现在是类 Taxi 的下属成员。在函数体中访问 totalFare 时不必加前缀“Taxi ::”，因为这时 AddTotal 和 totalFare 都在类 Taxi 中。类成员之间互相访问直接使用成员名，不需要加类名前缀。

静态函数成员的语法细则：

（1）声明时使用关键字 static 进行限定，定义时不能再使用关键字 static。

（2）类中的其他函数成员可以调用静态函数成员。调用时直接使用函数名，并且不受访问权限约束。这一点与调用普通函数成员是一样的。

（3）在类外调用静态函数成员需以“类名 :: 静态函数成员名( )”的形式调用，或通过任何一个该类对象以“对象名.静态函数成员名( )”的形式调用，或通过任何一个该类对象指针以“对象指针名->静态函数成员名( )”的形式调用。

类外调用受访问权限约束，只能调用公有的静态函数成员。访问权限决定了静态函数成员的作用域。私有静态函数成员具有类作用域，只能在类内调用。公有静态函数成员具有文件作用域，可以被本文件中的任何函数调用，并且可通过类声明将其作用域扩展到任何程序文件。

（4）静态函数成员只能访问类中的静态数据成员，换句话说就是不能访问类中的非静态数据成员。因为静态函数成员可以在没有定义对象的情况下直接调用，而非静态数据成员在定义对象之前没有分配内存空间，不能访问。同样的原因，静态函数成员只能调用类中的其他静态函数成员，不能调用非静态函数成员。

（5）静态函数成员不能是内联函数，因为编译器在编译时会调整内联函数，可能会用到静态数据成员中的数据，但此时静态数据成员还未初始化，所访问到的数据是错误的。

### 3. 静态成员的应用

在类中定义静态成员的目的有两个，一是以类的形式管理全局变量和外部函数，这样可以更好地组织程序代码；二是将具有相同属性值的成员定义成静态数据成员，可以减少内存占用。

### 1）以类的形式管理全局变量和外部函数

程序员可以将全局变量或外部函数划归到某个具有关联关系的类中，作为类的静态成员与其他成员一起进行管理。对于一些通用的、不好归类的全局变量或外部函数，也可以将它们作为静态成员，从形式上归为一类进行统一管理。例如，可以定义一个数学类 Math 统一管理常用的数学函数。

```
class Math
{
public:
        static double PI;
        static double sin( double x );
        static double cos( double x );
        …
} ;
double Math :: PI = 3.1415926;
double Math :: sin( double x )   {   …   };
double Math :: cos( double x )   {   …   };
…
```

以静态成员的形式访问 Math 类中的成员，例如：

```
cout << Math :: PI << endl;                  // 显示 Math 类中静态数据成员 PI 的值
cout << Math :: sin（2）<< endl;              // 调用 Math 类中的静态函数成员 sin
```

本质上，静态数据成员就是一个全局变量，静态函数成员就是一个外部函数。将它们改为静态成员的好处是便于分类组织和管理程序代码。

### 2）将具有相同属性值的成员定义成静态数据成员

仔细分析一下图 7-12 中对象数组 tObj 各数组元素的数据成员 price，它被用于保存每辆出租车的里程单价。如果所有出租车的里程单价都一样（即具有相同的属性值），那么还有必要为每个数组元素都分配一个保存里程单价的内存单元吗？答案是否定的——没有必要。

利用静态数据成员共用内存单元的特点，可以将具有相同属性值的成员定义成静态数据成员，这样可以有效减少内存占用。例如，修改例 7-12 中出租车类 Taxi 的定义代码，将数据成员 price 改为静态成员。

```
class Taxi                              // 定义一个出租车类 Taxi
{
private:
        static int   price;             // 修改 1：将里程单价修改成静态数据成员
        int   fare;                     // 数据成员：保存收费总额
public:
        //此处代码省略
        Taxi(int p=0, int f=0)          // 修改 2：静态数据成员在类实现部分定义并初始化
      {   price = p;   fare = f;   }
        static int   totalFare;         // 静态数据成员：公司总收费金额
};
int   Taxi :: price = GetARand(2, 5);   // 修改 3：定义并初始化静态数据成员 price
```

```
int   Taxi :: totalFare = 0;            // 定义并初始化静态数据成员 totalFare
```

使用修改后新的出租车类 Taxi 定义对象数组 tObj：

```
Taxi   tObj[10];                        // 定义新 Taxi 类的对象数组，可保存 10 辆出租车的数据
```

图 7-13 给出了这个新对象数组 tObj 的内存分配示意图。其中所有数组元素共用同一个里程单价，即 Taxi::price。

| | | 各数据成员在出租车类 Taxi 中的定义 |
|---|---|---|
| 静态存储区：Taxi::totalFare | 4 字节 | static int totalFare; |
| Taxi::price | 4 字节 | static int price; |
| 以下为自动存储区 | …… | |
| tObj[0] | 4 字节 | int fare; |
| …… | 4 字节 | int fare; |
| tObj[9] | 4 字节 | Int fare; |

图 7-13　新对象数组 tObj 的内存分配示意图

因为将数据成员 price 改成静态成员，需要同步对 Taxi 类的构造函数进行修改，这就是程序代码的关联修改。例 7-12 中还有 2 处程序代码需要做关联修改。

（1）主函数中不再需要设置每辆出租车的里程单价。

（2）可以将 Taxi 类中与静态数据成员 price 相关的函数成员 SetPrice 和 GetPrice 改为静态函数成员。

读者可根据上述线索，自行完成对例 7-12 程序代码的修改。

## 本节习题

1. 下列关于常成员的描述中，错误的是（　　）。
   A. 在类中声明常成员时需使用关键字 const
   B. 常数据成员需在声明时直接初始化
   C. 常函数成员只能读类中的数据成员，不能赋值修改
   D. 常函数成员只能调用其他常函数成员
2. 在类中声明一个常数据成员 x，下列哪条语句是正确的？（　　）
   A. int x;　　B. const int x;　　C. int const x;　　D. int x const;
3. 在类中声明一个常函数成员 fun，下列哪条语句是正确的？（　　）
   A. void fun( );　　B. const void fun( );
   C. void const fun( );　　D. void fun( ) const;
4. 下列关于静态成员的描述中，错误的是（　　）。
   A. 静态数据成员不属于某个对象，它是类的共享成员
   B. 静态数据成员要在类外定义和初始化
   C. 公有静态成员具有文件作用域

D. 私有静态成员具有块作用域

5. 已定义类 A：

```
class A
{
public:
    int x;
    static int y;
};
int A::y = 0;
```

下列语句中错误的是（　　）。

A. A a;　a.x = 1;　a.y = 1;　　　B. A a;　a.x = 1;　A::y = 1;

C. A::x = 1;　A::y = 1;　　　D. A::y = 1;

## 7.7 类的友元

友元是一个与访问权限相关的概念。编写类的程序员在定义类的时候为类成员赋予不同的访问权限。将必须被外部访问的成员开放出来，以保证类的功能可以被正常使用。将不需要被外部访问的成员隐藏出来，以防它们被误访问。被隐藏的成员只在内部使用，能被类里的其他成员访问，但不能在类的外部访问。

假设有一个类 A，另外还有两个类外的函数 fun1 和 fun2：

```
class A
{
public:     int x;                  // 公有成员 x
protected:  int y;                  // 保护成员 y
private:    int z;                  // 私有成员 z
public:
    A(int p1=0, int p2=0, int p3=0)  {  x = p1;   y = p2;   z = p3;  }  // 构造函数
} ;
void fun1( )
{
    A obj1( 2, 4, 6 );              // 定义 A 类对象 obj1，将其 3 个成员分别初始化为 2、4、6
    cout << obj1.x << endl;         // 正确：可以访问对象的公有成员 x
    cout << obj1.y << endl;         // 错误：不能访问对象的保护成员 y
    cout << obj1.z << endl;         // 错误：不能访问对象的私有成员 z
}
void fun2( )
{
    A obj2( 3, 5, 7 );              // 定义 A 类对象 obj2，将其 3 个成员分别初始化为 3、5、7
    cout << obj2.x << endl;         // 正确：可以访问对象的公有成员 x
    cout << obj2.y << endl;         // 错误：不能访问对象的保护成员 y
    cout << obj2.z << endl;         // 错误：不能访问对象的私有成员 z
}
```

类 A 中有 3 个成员，分别为公有成员 x、保护成员 y 和私有成员 z。函数 fun1、fun2 分别定义了一个 A 类对象 obj1 和 obj2。无论在哪个函数中访问 A 类对象的成员，都只能访问公有成员（例如 x），不能访问保护成员或私有成员（例如 y、z）。类中设定的访问权限对类外的所有函数一视同仁，具有相同的约束力。公有成员对大家都开放，都能访问。保护成员和私有成员是隐藏的，对大家都不开放，不能访问。

能否向类外某些函数定向开放类中隐藏的成员，实现更加精细化的类成员访问控制？例如程序员在定义类 A 的时候是否可以单独授权函数 fun1 访问类中的私有成员？答案是肯定的。C++语言可以在定义类的时候声明友元，向类外的某些函数或类定向开放类中的所有成员。被类声明为友元的函数称为该类的**友元函数**，被声明为友元的类称为该类的**友元类**。

## 7.7.1 友元函数

C++语法：在类中声明友元函数

```
class 类名        // 类声明部分
{
    …
    friend  友元函数的原型声明 ;
} ;
```

语法说明：

- 在**类声明部分**声明友元函数的原型，声明时使用关键字 **friend**。声明语句可以放在大括号内的任意位置，该位置的访问权限与友元函数无关。
- **友元函数**是类外的其他函数，不是类的成员。
- **友元函数**可以在其函数体内访问该类对象的所有成员，不受权限约束。

例如，如果将上述函数 fun1 声明为类 A 的友元函数：

```
class A
{
public:      int x;                          // 公有成员 x
protected:   int y;                          // 保护成员 y
private:     int z;                          // 私有成员 z
public:
    A(int p1=0, int p2=0, int p3=0)  {  x = p1;   y = p2;   z = p3;  }  // 构造函数
    friend   void fun1( );                   // 声明函数 fun1 为类 A 的友元函数
} ;
```

则在函数 fun1 的函数体中可访问 A 类对象的所有成员：

```
void fun1( )
{
    A obj1( 2, 4, 6 );              // 定义 A 类对象 obj1，将其 3 个成员分别初始化为 2、4、6
    cout << obj1.x << endl;         // 正确：可以访问对象的公有成员 x
    cout << obj1.y << endl;         // 正确：友元函数可以访问对象的保护成员 y
```

```
    cout << obj1.z << endl;          // 正确：友元函数可以访问对象的私有成员 z
}
```

而函数 fun2 仍然不能访问 A 类对象的保护成员和私有成员，只能访问其公有成员。

## 7.7.2　友元类

类 A 的友元函数可以是另一个类 B 的函数成员。假设上述函数 fun1 和 fun2 是类 B 的函数成员：

```
class B
{
    …
    void fun1( )  {  …  }
    void fun2( )  {  …  }
} ;
```

则在类 A 中声明友元函数 fun1 时需在函数名前面加上类名限定：

```
class A
{
    …
    friend   void B :: fun1( );       // 声明类 B 的函数成员 fun1 为类 A 的友元函数
} ;
```

如果类 B 的所有函数成员都是类 A 的友元函数，则称类 B 为类 A 的**友元类**。声明友元类时不需要逐个声明其函数成员，而是采用统一声明的语法形式：

```
class A
{
    …
    friend   class B;                // 声明类 B 为类 A 的友元类
} ;
```

声明为类 A 的友元类后，类 B 的函数成员就都可以在函数体中访问 A 类对象的所有成员，包括保护成员和私有成员。

应用友元类时需注意两点：

（1）**友元关系是单向的**。若类 A 声明类 B 是自己的友元，并不意味着 A 同时成为 B 的友元，除非对方声明。

（2）**友元关系不能传递**。假设类 B 是类 A 的友元，类 C 又是类 B 的友元，这并不意味着类 A 和 C 之间存在任何友元关系，除非它们自己单独声明。

## 本节习题

1. 下列关于友元函数的描述中，错误的是（　　）。
   A. 在外部函数中访问某个对象的成员时，只能访问对象的公有成员
   B. 如果函数是类的友元函数，则在该函数中就可以访问该类对象的私有成员

C. 在类定义中声明友元函数，需要使用关键字 friend

D. 类的友元函数是一种属于该类的特殊函数成员

2. 下列友元函数的定义中，错误的是（　　）。

A.
```
class ABC
{
    private: int x, y;
    public: ABC(int a=0, int b=0)  {  x = a;  y = b;  }
    friend void Show(ABC obj);
};
void Show(ABC obj)  {  cout << obj.x << obj.y;  }
```

B.
```
class ABC
{
    friend void Show( ABC );
    private: int x, y;
    public: ABC(int a=0, int b=0)  {  x = a;  y = b;  }
};
void Show(ABC obj)  {  cout << obj.x << obj.y;  }
```

C.
```
class ABC
{
    private: int x, y;
    public: ABC(int a=0, int b=0)  {  x = a;  y = b;  }
    friend Show( );
};
void Show(ABC obj)  {  cout << obj.x << obj.y;  }
```

D.
```
class ABC
{
    private: int x, y;
    public:
    ABC(int a=0, int b=0)  {  x = a;  y = b;  }
    friend void Show(ABC obj)  {  cout << obj.x << obj.y;  }
};
```

3. 如需将类 B 的函数成员“void fun( );”定义成类 A 的友元函数，则需在类 A 中增加下列哪条语句？（　　）

A. void fun( );　　　　B. friend void fun( );

C. friend void A::fun( );　　D. friend void B::fun( );

4. 下列关于友元类的描述中，错误的是（　　）。

A. 如果希望类 B 的函数成员都是类 A 的友元函数，则可将类 B 定义成类 A 的友元类

B. 将类 B 定义成类 A 友元类的方法是在类 A 中增加一条如下的声明语句：

```
friend class B;
```

C. 如果类 B 是类 A 的友元类，那么类 A 自动成为类 B 的友元类
D. 如果类 B 是类 A 的友元类，类 C 又是类 B 的友元类，此时 C 不一定是 A 的友元类

### 学习本章的要点

- 读者必须从代码分类管理、数据类型、归纳抽象和代码重用等多个维度才能准确理解类与对象的概念。
- 读者需认真学习类与对象编程的具体语法规则。
- 读者要深入领会面向对象程序设计通过设置访问权限来实现类封装的基本思想。
- 读者要深入了解对象的构造与析构过程。程序员通过编写构造与析构函数来参与对象的构造与析构过程。
- 读者要从两个不同的角色，即定义类的程序员和使用类定义对象的程序员，才能更容易地理解类与对象相关的各种语法知识。

## 7.8　本章习题

1. **阅读程序**。阅读下列 C++程序。阅读后请说明程序的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
using namespace std;
class CTest
{
private:    int x, y;
public:
        void Set(int p1, int p2);
        int GetX( )    {    return x;    }
        int GetY( )    {    return y;    }
};
void CTest::Set(int p1, int p2)    {    x = p1;    y = p2;    }
int main( )
{
        CTest obj1, obj2;
        obj1.Set(2, 5);
        obj2.Set(3, 6);
        cout << obj1.GetX( ) * obj1.GetY( ) << endl;
        cout << obj1.GetX( ) * obj2.GetY( ) << endl;
        cout << obj1.GetY( ) * obj2.GetX( ) << endl;
        cout << obj2.GetX( ) * obj2.GetY( ) << endl;
        return 0;
}
```

2. **阅读程序**。阅读下列 C++程序。阅读后请说明程序的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
using namespace std;
class CTest
{
private:    int x, y;
public:
        CTest(int p1 = 0, int p2 = 0)    {    x = p1;    y = p2;    }
        CTest(CTest &p)    {    x = p.x;    y = p.y;    }
        void Show( )    {    cout << x << ", " << y << endl;    }
};
int main( )
{
        CTest obj1;
        obj1.Show( );
        CTest obj2(2, 5);
        obj2.Show( );
        CTest obj3(obj2);
        obj3.Show( );
        return 0;
}
```

3. **阅读程序**。阅读下列 C++程序。阅读后请说明程序的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
using namespace std;
class Line
{
private:
        int length;
        void Show( )
        {    for (int n=0; n < length; n++)    cout << '-';    }
public:
        Line(int a=0);
        ~Line( );
        void Longer( );
        void Shorter( );
} ;
Line :: Line(int a)
{    length = a;    Show( );    cout << "    我出生了\n\n";    }
Line :: ~Line( )
{    cout << "我消失了!\n";    }
void Line :: Longer( )
{    length *= 2;    Show( );    cout << "    我变长了\n\n";    }
void Line :: Shorter( )
{    length /= 2;    Show( );    cout << "    我变短了\n\n";    }
int main( )
```

```
{
    char choice;
    Line obj( 8 );
    while (true)
    {
        cin >> choice;
        if (choice == 'L' || choice == 'l')   obj.Longer( );
        else if (choice == 'S' || choice == 's')   obj.Shorter( );
        else   break;
    }
    return 0;
}
```

4. **编写程序**。编写一个关于圆形的 C++程序。要求定义一个圆形类 Circle，其中包含如下成员：

（1）1 个私有数据成员（半径）。

（2）3 个公有函数成员（设置半径、计算面积和周长）。

（3）3 个构造函数（不带参数的构造函数、带参数的构造函数和拷贝构造函数）。

主函数 main 使用圆形类 Circle 定义圆形对象，要求：

（1）定义一个圆对象 c1，然后从键盘输入一个数值 x 并将其设定为 c1 的半径，计算并显示 c1 的面积和周长。

（2）再定义一个圆对象 c2 并将半径初始化为 2x，计算并显示 c2 的面积和周长。

（3）再定义一个圆对象 c3 并用 c1 初始化 c3，计算并显示 c3 的面积和周长。

5. **编写程序**。设计一个带日历的钟表类 Clock。编写类定义代码并测试这个类。

6. **编写程序**。设计一个关于人事管理的类 Employee。要求类 Employee 包含员工编号、姓名、部门和工资等数据成员；对员工进行操作的函数成员，例如设置员工信息、增加和显示工资等。编写类定义代码并测试这个类。

# 第8章 面向对象程序设计之二

面向对象程序设计之所以能有效提高程序开发效率，其主要的技术手段有两个，一是分类管理程序代码，二是重用类代码。上一章已讲解了如何分类管理程序代码，即**类与对象**编程。C++语言经过三十多年的发展，已经积累了大量编写好的具有各种不同功能的类。重用类代码，就是利用已有的类来编写程序，这样可以快速开发程序。本章将介绍如何重用类代码，重点讲解类的**组合**与**继承**。

本章还会深入讲解面向对象程序设计方法中的另外一个重要思想，即**多态**。多态性在字面上可理解为是一种程序代码的多义性。面向对象程序设计之所以提出多态的思想，其目的仍然是为进一步提高程序代码的可重用性，进而提高软件开发和维护效率。

## 8.1 重用类代码

“程序=数据+算法”。程序中的数据包括原始数据、中间结果和最终结果等。如何根据所处理的数据来合理使用和管理内存是编写程序的第一项工作内容。C++语言通过定义变量语句来申请内存空间。定义变量语句是与数据相关的代码，即**数据代码**。

将数据处理的过程细分成一组严格的操作步骤，这组操作步骤被称为算法。如何设计数据处理算法是编写程序的第二项工作内容。C++语言通过定义函数来描述算法模块。函数是与算法相关的代码，即**算法代码**。

在面向对象程序设计中，类是重用“数据代码+算法代码”的基本语法形式。重用已有的类代码，可以同时重用其中的数据代码和算法代码，实现了对已有程序代码的完全重用，这极大地提高了程序开发的效率。目前，面向对象程序设计方法是主流。

面向对象程序设计重用类代码有三种形式，分别是用类定义对象、通过组合来定义新类或通过继承来定义新类。

### 8.1.1 用类定义对象

在面向对象程序设计中，程序员将相对独立、经常使用的功能提炼出来，编写成类这样可以重用的代码形式。假设程序员甲将与圆形相关的程序功能提炼出来，用 C++语言编写成一个圆形类 Circle。例 8-1 给出了一个完整的圆形类 Circle 的定义代码。

**例 8-1　圆形类 Circle 的定义代码（程序员甲编写）**

类声明头文件：Circle.h

```cpp
class Circle        // 圆形类：声明部分
{
private:
    double r;   // 半径：私有数据成员

public:
    void Input( );          // 输入半径
    double Radius( );       // 读取半径
    double CArea( );        // 求圆形面积
    double CLen( );         // 求圆形周长
    // 以下 3 个构造函数被定义成内联函数
    Circle( )   // 无参构造函数
    {    r = 0;    }
    Circle( double x )              // 有参构造函数
    {
        if (x < 0)   r = 0;         // 数据合法性检查
        else   r = x;
    }
    Circle( Circle &x )             // 拷贝构造函数
    {    r = x.r;    }
};
```

类实现程序文件：Circle.cpp

```cpp
#include <iostream>
using namespace std;
#include "Circle.h"   // 声明类 Circle

// 圆形类：实现部分
void Circle :: Input( )     // 输入半径
{
    cin >> r;
    while ( r < 0 )   // 数据合法性检查
        cin >> r;   // 如 r<0 则重新输入
}
double Circle :: Radius( )   // 读取半径
{    return   r;    }
double Circle :: CArea( )   // 求圆形面积
{    return (3.14*r*r);    }
double Circle :: CLen( )   // 求圆形周长
{    return (3.14*2*r);    }
```

一个完整的类定义应包括 5 大要素，即数据成员、函数成员、各成员的访问权限、构造函数和析构函数。程序员甲将 Circle 类的定义代码分成 2 个文件，一个是类声明头文件 Circle.h，另一个是类实现程序文件 Circle.cpp。

为了保证数据合法性，类 Circle 将数据成员半径 r 隐藏起来，设置为私有成员。为了让使用类的程序员能够设置对象的半径，类 Circle 增加一个输入半径的公有函数成员 Input，该函数对从键盘输入的数据进行合法性检查。另外还增加一个读取半径的公有函数成员 Radius。

假设程序员乙想编写一个计算圆形面积和周长的程序，那么就可以使用 Circle 类定义对象，然后通过对象重用 Circle 类的代码。一个使用类 Circle 的典型流程如下：

```cpp
Circle   obj;                        // 定义一个 Circle 类的对象 obj
obj.Input( );                        // 调用对象 obj 的公有函数成员 Input，输入其半径
cout << obj.Radius( ) << endl;       // 调用对象 obj 的公有函数成员 Radius，读取并显示半径
cout << obj.CArea( ) << endl;        // 调用对象 obj 的公有函数成员 CArea，计算并显示面积
cout << obj.CLen( ) << endl;         // 调用对象 obj 的公有函数成员 CLen，计算并显示周长
```

类 Circle 还定义了 3 个重载的构造函数（在本例中被定义成内联函数），分别是无参构造函数、有参构造函数和拷贝构造函数，这样可以为 Circle 类的对象提供 3 种不同的初始化方式，例如：

```
Circle   obj1;              // 自动调用无参构造函数，将对象 obj1 的半径初始化成 0
Circle   obj2( 5 );         // 自动调用有参构造函数，将对象 obj2 的半径初始化成 5
Circle   obj3( obj2 );      // 自动调用拷贝构造函数，将对象 obj3 的半径也初始化成 5
```

类 Circle 没有定义析构函数，编译器在编译时将会自动添加一个默认析构函数，其语法形式为：

```
~Circle( )   {   }                  // 空函数，什么事情都没做
```

用类定义对象是重用类代码的第一种形式（见图 8-1）。在这个重用过程中存在两个程序员角色，程序员甲定义类，编写类代码；程序员乙使用类定义对象，然后通过对象重用类代码。

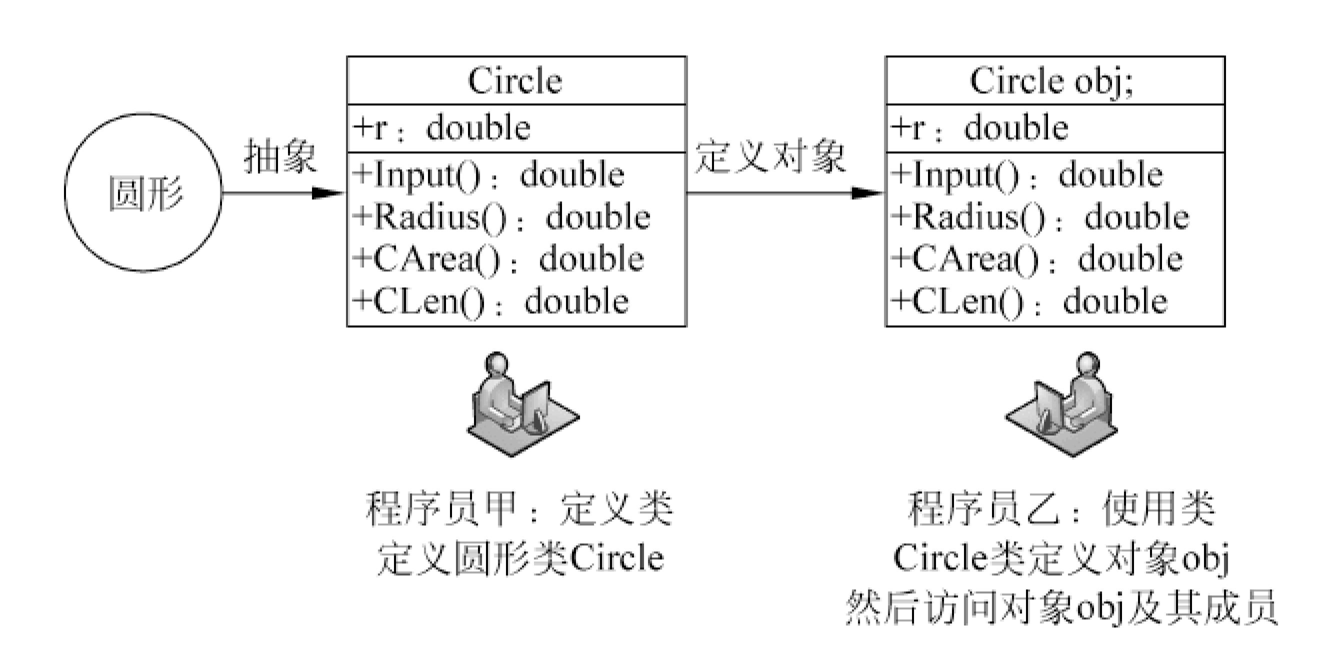



图 8-1　用类定义对象

## 8.1.2　用类继续定义新类

设想这样一个编程场景：假设需要编写一个处理圆柱体的 C++程序。程序由乙和丙两位程序员分工协作，共同编写。采用面向对象程序设计方法，程序员乙负责编写圆柱体类 Cylinder，程序员丙负责编写主函数 main。

程序员乙经过分析可抽象出圆柱体的数据模型，其中应包含 2 个数据成员（圆形底面的半径 r 和圆柱体的高度 h），还应包含 5 个函数成员（输入半径和高度的函数 Input、求圆形底面积的函数 CArea、求圆形底面周长的函数 CLen、求表面积的函数 Surface 和求体积的函数 Volumn 等）。

程序员乙按照数据模型编写圆柱体类 Cylinder 的定义代码，然后程序员丙在主函数中用 Cylinder 类定义对象，再通过对象重用类代码，最终完成对圆柱体的处理功能。图 8-2 给出了上述编程过程的示意图。这里程序员丙也是通过定义对象来重用 Cylinder 类的代码，重用形式与图 8-1 一样。

下面我们将重点聚焦到程序员乙的身上，看看他在定义圆柱体类 Cylinder 时有没有什么更好的方法。图 8-2 中，程序员乙所定义的圆柱体类 Cylinder 是从零开始编写的。仔细分析一下，圆柱体数据模型包含一个圆形底面和一个高度。在例 8-1 中，程序员甲已经建立了圆形数据模型，并用 C++语言定义了一个圆形类 Circle。程序员乙可以借助已有的圆形类 Circle 来定义新的圆柱体类 Cylinder（见图 8-3）。

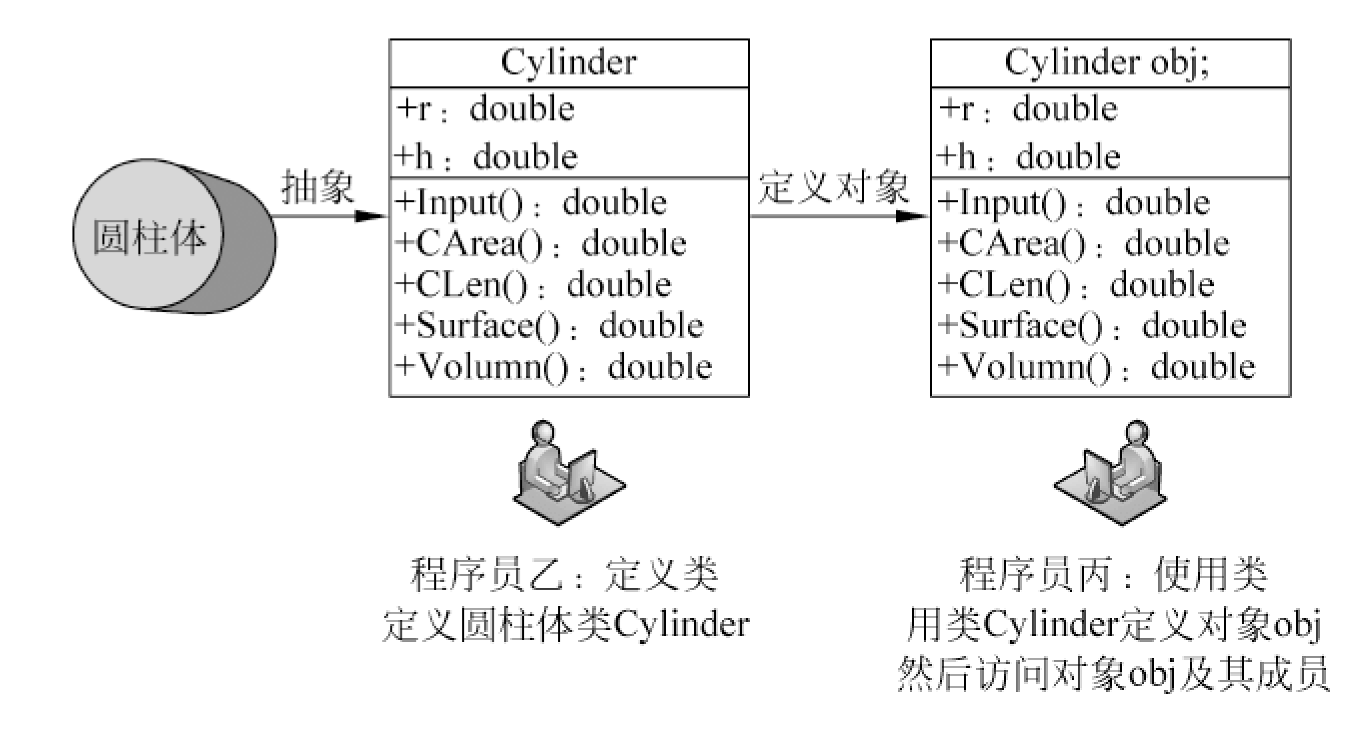



图 8-2　圆柱体处理程序的编程过程

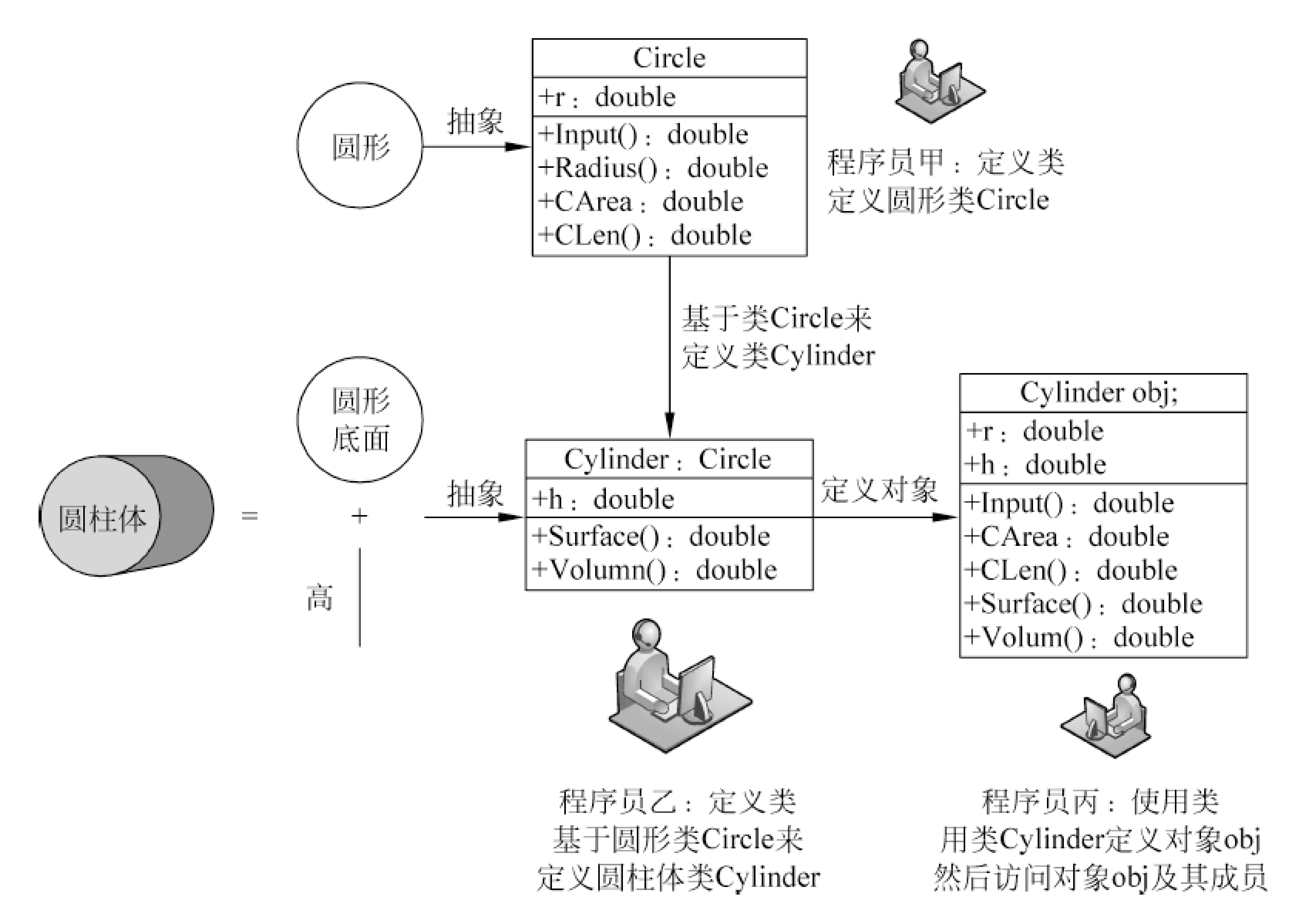



图 8-3　用类继续定义新类

程序员乙基于圆形类 Circle 来定义新的圆柱体类 Cylinder，定义时不需要再重复定义数据成员半径 r、求圆形面积函数 CArea 和求圆形周长函数 CLen，只需要补充与圆柱体相关的新内容，例如高度 h、求表面积函数 Surface 和求体积函数 Volumn 等。

用已有的类定义新类，就是重用已有类的代码，这样可以提高新类的开发效率。这时在重用类代码过程中会涉及三个程序员角色，程序员甲是编写已有类（例如图 8-3 中的类 Circle）的程序员，程序员乙是编写新类（例如图 8-3 中的类 Cylinder）的程序员，程序员丙则是使用新类定义对象（例如图 8-3 中的 Cylinder 类对象 obj）的程序员。

用已有的类定义新类，可以采用组合或继承两种不同的方法。通过组合所定义出的新类被称为组合类，通过继承所定义出的新类被称为派生类。下一节开始我们将深入学习类的组合和继承方法，具体内容包括：

（1）组合和继承的编程原理。

（2）组合类和派生类的定义语法。

（3）组合类或派生类可以定义对象，所定义出的组合类或派生类对象有一些特殊之处。

程序员乙的角色需要学习(1)和(2)，这样才能用已有的类来定义新类。程序员丙的角色需要学习(3)，学会如何访问组合类对象或派生类对象。

### 本节习题

1. 计算机程序由哪两个基本要素组成？（　　）

A. 程序和程序员　　B. 软件和硬件　　C. 数据和算法　　D. 类和对象

2. 结构化程序设计中调用函数，重用的是什么？（　　）

A. 程序员　　B. 数据代码

C. 算法代码　　D. 数据代码+算法代码

3. 结构化程序设计中使用结构体定义变量，重用的是什么？（　　）

A. 程序员　　B. 数据代码

C. 算法代码　　D. 数据代码+算法代码

4. 面向对象程序设计中使用类定义对象，重用的是什么？（　　）

A. 程序员　　B. 数据代码

C. 算法代码　　D. 数据代码+算法代码

5. 面向对象程序设计中重用类代码的形式不包括下列哪一种？（　　）

A. 用类定义对象　　B. 类的组合

C. 类的继承　　D. 拷贝类代码

## 8.2 类的组合

类不是 C++语言预定义的基本数据类型，而是由多个基本类型的数据成员组合在一起形成的自定义数据类型。用简单的零件组装复杂的整体是人们常用的一种方法，例如计算机工厂将主板、CPU、内存条、硬盘等零件组装在一起生产出整机。零件通常不是自己生产，而是从专门厂家购买来的，这样可以降低生产难度，提高效率。

程序员可以将别人编写的类当作零件(**零件类**)，在此基础上定义自己的新类(**整体类**)，这就是类的组合。组合的**编程原理**是：程序员在定义新类的时候，使用已有的类来定义数据成员。这些数据成员是类类型的对象，被称为**对象成员**。C++语言将数据成员中包含对象成员的类称为**组合类**。按照数据类型的不同，组合类中数据成员可分为两种，即类类型的对象成员和基本数据类型的非对象成员。

使用组合类定义对象，即组合类对象，其成员中也将包含对象成员和非对象成员。访问组合类对象中的**非对象成员**，其访问形式与之前所介绍的访问数据成员没有区别，即：

**组合类对象名 . 非对象成员名**

而组合类对象中的对象成员还包含自己的下级成员，即组合类对象包含**多级成员**。可

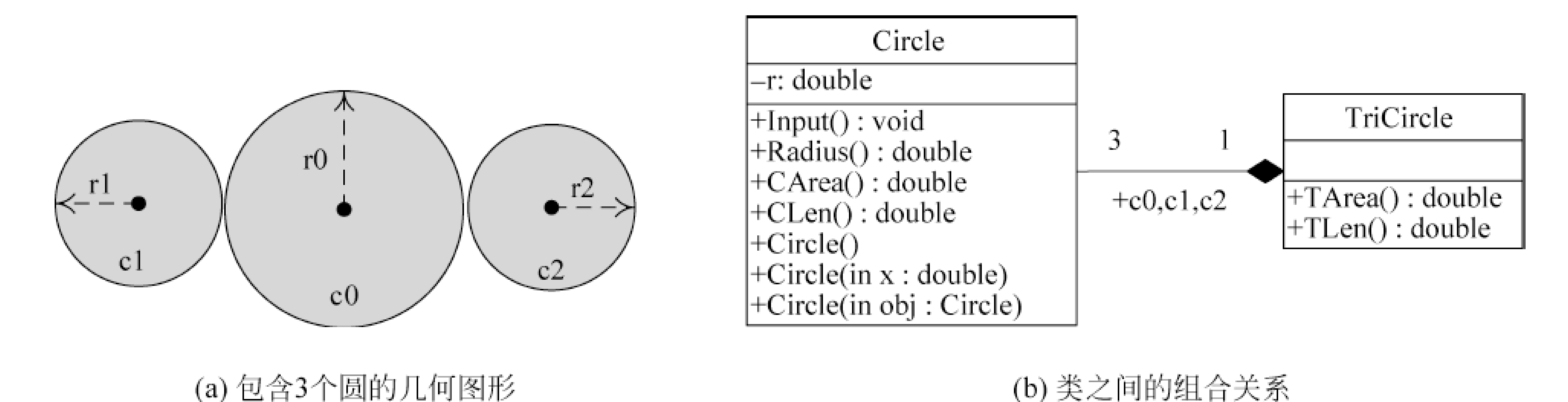
以访问组合类对象中对象成员的下级成员，这是一种**多级访问**。多级访问的语法形式是：

**组合类对象名 . 对象成员名 . 对象成员的下级成员名**

请注意：多级访问将受到**多级权限**的控制。

## 8.2.1　组合类的定义

假设我们要定义一个描述图 8-4(a)中几何图形的类 TriCircle。图中所示的几何图形包含 3 个圆 c0、c1、c2，其半径分别为 r0、r1、r2，对该几何图形的处理可能包括计算各个圆的面积、周长，以及计算其总面积和总周长等。




(a) 包含3个圆的几何图形　　(b) 类之间的组合关系

图 8-4　组合类举例

定义 TriCircle 类可以从零开始编写，也可以基于例 8-1 的 Circle 类来编写组合类，后者的开发效率更高。类 TriCircle 可以认为是由 3 个 Circle 类对象组合而成的，图 8-4(b)用 UML（统一建模语言）描述了这两个类之间的组合关系。基于 Circle 类来定义组合类 TriCircle，其定义代码如例 8-2 所示。

**例 8-2　组合类 TriCircle 的定义代码**

**类声明头文件：TriCircle.h**

```cpp
#include "Circle.h"   // 声明类 Circle

class TriCircle   // 声明部分，即声明成员
{
public:
    // 声明 3 个公有的 Circle 类对象成员
    Circle c0, c1, c2;

    // 以下为 TriCircle 类相关的新内容
    double TArea( );   // 求总面积：公有函数成员
    double TLen( );   // 求总周长：公有函数成员
};
```

**类实现程序文件：TriCircle.cpp**

```cpp
#include "TriCircle.h"   // 声明类 TriCircle

// TriCircle 类：实现部分，具体的函数代码
double TriCircle :: TArea( )   // 求总面积
{
    double totalArea;
    totalArea = c0.CArea( ) +
                c1.CArea( ) + c2.CArea( );
    return   totalArea;
}
double TriCircle :: TLen( )   // 求总周长
{
    double totalLen;
    totalLen = c0.CLen ( ) +
               c1.CLen( ) + c2.CLen( );
    return   totalLen;
}
```

例 8-2 的程序说明如下。

1) **组合类 TriCircle 中的对象成员**

例 8-2 所定义的组合类 TriCircle 中声明了 3 个数据成员 c0、c1、c2。它们都是 Circle 类的对象，被称为是组合类 TriCircle 的对象成员。对象成员中还包含下级成员，也就是说组合类包含多级成员。重用 Circle 类的代码，这样组合类 TriCircle 就不需要再重复定义数据成员半径 r、输入半径函数 Input、求圆形面积函数 CArea 和求圆形周长函数 CLen，只需要补充与 TriCircle 类相关的新内容，例如补充声明了 2 个新的函数成员 TArea 和 TLen，TArea 的功能是求总面积，TLen 的功能是求总周长。

2) **在组合类中访问对象成员**

组合类 TriCircle 中函数成员 TArea 通过调用对象成员 c0、c1、c2 的下级函数成员 CArea 来计算总面积。在组合类中访问对象成员通常是访问其下级成员，访问时受下级成员的访问权限控制。对象成员中，只有公有权限的下级成员才能被访问。

3) **组合类中对象成员的二次封装**

组合类 TriCircle 将对象成员 c0、c1、c2 的访问权限设定成 public。组合类设定对象成员的访问权限实际上是对它们的二次封装。将 c0、c1、c2 设定成 public 就是开放这 3 个对象成员，如图 8-5(a)所示。如果二次封装时将对象成员设定成 private，则对象成员及其下级成员都将被隐藏起来，如图 8-5(b)所示。此时，这些对象成员及其下级成员只能在组合类的内部才能访问，在类的外部不能访问。

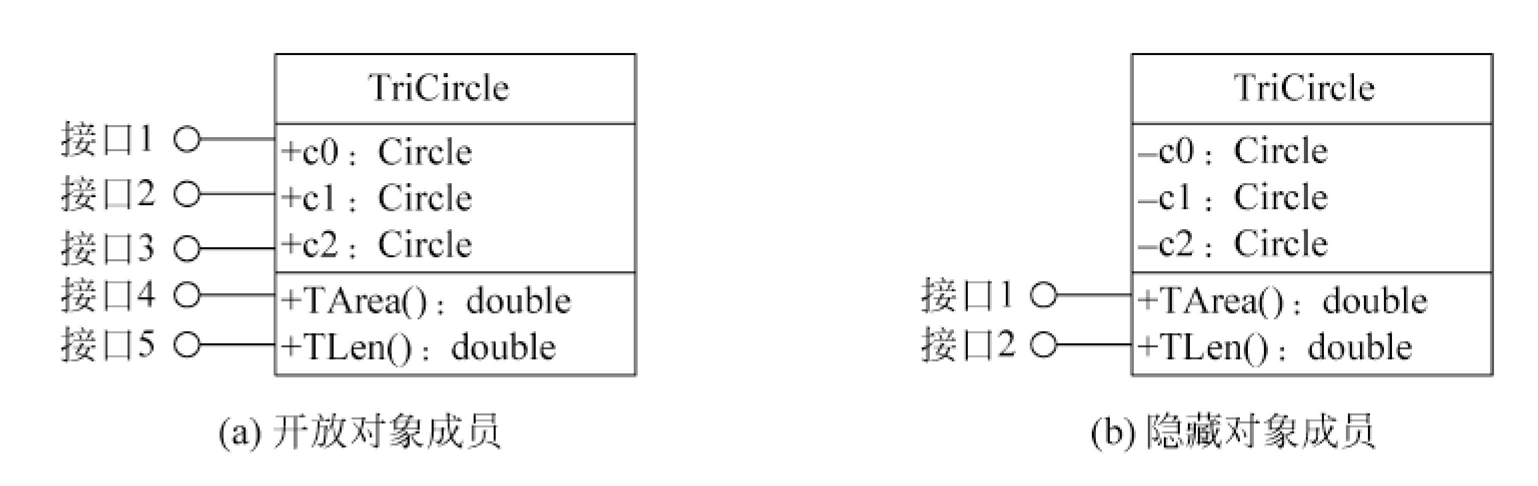



图 8-5　组合类中对象成员的二次封装

## 8.2.2　组合类对象的定义与访问

与任何普通的类一样，可以使用组合类来定义对象。例如，定义一个组合类 TriCircle 的对象 obj：

```
TriCircle   obj;        // 定义一个组合类 TriCircle 的对象 obj
```

计算机执行该对象定义语句时将为对象 obj 分配内存空间（如图 8-6 所示）。一个组合类对象所占用的内存空间等于类中全部数据成员（含对象成员）所需内存空间的总和。

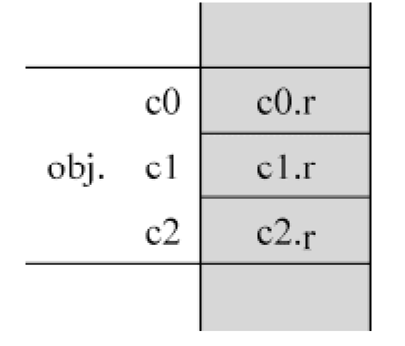



图 8-6　组合类对象 obj 的内存示意图

### 1. 组合类对象中对象成员的多级访问

按照 TriCircle 类的定义，对象 obj 将包含 3 个

数据成员（c0、c1、c2）和 2 个函数成员（TArea、TLen）。这 5 个成员都是公有成员，可以访问，其访问形式分别是：

**obj.c0、obj.c1、obj.c2、obj.TArea( )、obj.TLen( )**

与普通对象不同的是，组合类对象的数据成员中含有对象成员。对象成员还包含下级成员，可以访问这些下级成员中的公有成员，这就是组合类对象的多级访问。多级访问的语法形式是：

**组合类对象名 . 对象成员名 . 对象成员的下级成员名**

组合类对象 obj 中的 c0、c1 和 c2，它们都是 Circle 类的对象，都有各自的下级成员（1 个私有数据成员 r 和 4 个公有函数成员 Input、Radius、CArea、Clen）。可以访问对象成员 c0、c1 和 c2 的下级公有成员。例如，下面的主函数演示了如何利用组合类对象的多级访问来计算图 8-4(a)所示几何图形中各个圆的面积、周长，以及总面积和总周长。

```
#include <iostream>
using namespace std;
#include "TriCircle.h"                 // 类 TriCircle 的声明头文件
int main( )
{
      TriCircle   obj;                 // 定义一个组合类 TriCircle 的对象 obj
      // 调用组合类对象 obj 中对象成员 c0 的下级函数成员 Input，输入 c0 的半径
      obj.c0.Input( );
      // 再调用 c0 的下级函数成员 CArea 和 CLen，计算并显示 c0 的面积和周长
      cout << obj.c0.CArea( ) << ", " << obj.c0.CLen( ) << endl;

      // 类似地，可计算并显示出 obj 中对象成员 c1、c2 的面积和周长
      obj.c1.Input( );      cout << obj.c1.CArea( ) << ", " << obj.c1.CLen( ) << endl;
      obj.c2.Input( );      cout << obj.c2.CArea( ) << ", " << obj.c2.CLen( ) << endl;
      // 调用组合类对象 obj 中的非对象成员 TArea 和 TLen，计算并显示总面积和总周长
      cout << obj.TArea( ) << ", " << obj.TLen( ) << endl;
      return 0;
}
```

通过对象指针也可以间接访问组合类对象及其下级成员。例如，定义一个 TriCircle 类的对象指针 p，让其指向 TriCircle 类的对象 obj，然后就可以通过对象指针 p 间接访问组合类对象 obj 及其下级成员：

```
TriCircle   obj;              // 定义一个组合类 TriCircle 的对象 obj
TriCircle   *p = &obj;        // 定义一个组合类 TriCircle 的对象指针 p，让其指向 obj
// 通过对象指针 p 间接访问组合类对象 obj 中对象成员的下级成员
p->c0.Input( );      cout << p->c0.CArea( ) << ", " << p->c0.CLen( ) << endl;
p->c1.Input( );      cout << p->c1.CArea( ) << ", " << p->c1.CLen( ) << endl;
p->c2.Input( );      cout << p->c2.CArea( ) << ", " << p->c2.CLen( ) << endl;
// 通过对象指针 p 间接访问组合类对象 obj 中的非对象成员 TArea 和 TLen
cout << p->TArea( ) << ", " << p->TLen( ) << endl;
```

#### 2. 如何设计组合类中对象成员的访问权限

组合类将其他类（零件类）的对象作为自己的数据成员（即对象成员），相当于是用零件来组装产品。用零件组装产品时要考虑，是将零件直接暴露给用户，还是将零件隐藏起来。这个问题要根据产品及零件的功能来决定。例如，计算机厂家在组装计算机时根据功能要求，将用户不需要操作的主板、CPU、内存条和硬盘等零件隐藏起来，即用一个机箱将这些零件“封装”起来。将用户需要操作的电源开关、键盘、鼠标、光盘和显示器等零件开放出来，放在机箱外面。

在组合类 TriCircle 的定义和使用过程中有三个程序员角色，程序员甲是编写 Circle 类的程序员，程序员乙是编写组合类 TriCircle 的程序员，程序员丙则是使用 TriCircle 类定义组合类对象 obj 的程序员。

**程序员乙**使用 Circle 类的对象成员来组装组合类 TriCircle。组装组合类时，程序员乙应根据功能要求决定将哪些对象成员开放给程序员丙，哪些应当隐藏起来。开放就是将对象成员设定为公有权限，隐藏就是设定为保护权限或私有权限，这就是组合时对象成员的二次封装。例如在组装组合类 TriCircle 时，程序员乙根据功能要求决定将 3 个对象成员 c0、c1 和 c2 都开放给程序员丙，即将它们都设为公有权限。

**程序员丙**使用组合类定义对象，然后访问组合类对象及其下级成员。组合类对象包含对象成员，开放的对象成员可以访问，隐藏的则不可以访问。对象成员的访问权限是程序员乙在定义组合类时设定的。

对象成员还包含下级成员，程序员丙可以访问对象成员的下级成员，这就是组合类对象的多级访问。这些下级成员也都有各自的访问权限，它们是**程序员甲**在定义 Cirlce 类时设定的。程序员丙只能访问下级成员中的公有成员，多级访问受到多级权限的控制。

总结一下，程序员丙在访问组合类对象中对象成员的下级成员时，只有对象成员和下级成员都是公有权限才可以访问，否则就不能访问。

#### 3. 多级组合

使用零件组装出的产品可以继续作为零件，去组装更大的产品，组装可以任意多级。例如，计算机厂家用零件组装计算机，而 ATM 机厂家又会把计算机作为零件来组装 ATM 机。组装 ATM 机时会进行再次封装，例如将计算机的显示器继续开放出来，但把 ATM 机用户不需要的键盘、鼠标、光盘等接口都封装起来了。

组合类也可以任意多级。用零件类定义组合类，组合类可以继续作为零件类去定义更大的组合类，这就是**多级组合**。多级组合过程中，每一级组合类都会根据自己的功能需要设定对象成员的访问权限，这就产生了**多级封装**。

### 8.2.3 组合类对象的构造与析构

计算机执行定义对象语句时将创建对象，为其分配内存空间，并自动调用对象所属类的构造函数来初始化对象，这个过程就是对象的构造。当对象生存期结束时，计算机将销毁对象。销毁时自动调用对象所属类的析构函数来清理内存，然后释放其所占用的内存空间，这个过程就是对象的析构。

按照数据类型的不同，组合类中数据成员可分为两种，即类类型的对象成员和基本数据类型的非对象成员。初始化对象成员比较麻烦，因为对象成员的下级数据成员可能是私有的，不能直接访问赋值。

#### 1. 组合类的构造函数

构造函数通过形参传递初始值，实现对新建对象数据成员的初始化。组合类构造函数不能直接初始化类中的对象成员，因为对象成员的下级数据成员可能是私有的，不能访问赋值。要想初始化对象成员，必须通过其所属类的构造函数才能完成。调用对象成员所属类构造函数，其语法形式是在组合类构造函数的函数头后面添加**初始化列表**（粗体部分）：

```
组合类构造函数名( 形参列表 ) : 对象成员名 1( 形参 1 ), 对象成员名 2( 形参 2 ), …
{
    …  // 在函数体中初始化其他非对象成员
}
```

其中，形参 1、形参 2 等是从形参列表中提取出来的，并在初始化列表中进行二次接力传递。组合类对象中各数据成员的初始化顺序是：先调用对象成员所属类的构造函数，初始化对象成员；再执行组合类构造函数的函数体，初始化其他非对象成员。如果组合类中有多个对象成员，那么这些对象成员的初始化顺序由其在组合类中的声明顺序决定，先声明者先初始化。

例如，为组合类 TriCircle 添加如下 3 个重载构造函数：

（1）**有参构造函数**。为了初始化 3 个圆形对象成员的半径，需传递 3 个初始值。

```
iCircle :: TriCircle( double p0, double p1, double p2 ) : c0(p0), c1(p1), c2(p2)   {   }
```

**注意**：通过初始化列表初始化对象成员 c0、c1、c2。因为类 TriCircle 的数据成员中没有非对象成员，所以函数体为空。

（2）**无参构造函数**。

```
TriCircle :: TriCircle( )   {   }
```

（3）**拷贝构造函数**。接收一个已经存在的本类对象引用，用该对象来初始化新建对象。

```
TriCircle :: TriCircle( TriCircle &rObj ) : c0(rObj.c0), c1(rObj.c1), c2(rObj.c2)   {   }
```

上述 3 个构造函数为 TriCircle 类对象提供了三种不同的初始化方式，例如：

■ TriCircle　obj( 5, 2, 3 );

该定义语句给出 3 个实参（即初始值）。根据实参—形参匹配原则，执行该语句将自动调用有参构造函数(1)来初始化新建对象 obj。初始化列表分别将形参 p0、p1、p2 所接收到的实参数据接力传递给 Circle 类对应的有参构造函数，将对象成员 c0、c1、c2 的半径分别初始化为 5、2、3。

类 TriCircle 中 3 个对象成员的声明顺序如下：

```
Circle c0, c1, c2;                    // 声明 3 个公有 Circle 类对象成员
```

因此它们被初始化的顺序依次是 c0、c1、c2。

■ TriCircle obj1;

该定义语句没有给实参。根据实参—形参匹配原则，执行该语句将自动调用无参构造函数(2)来初始化新建对象 obj1。虽然无参构造函数没写初始化列表，但仍然会自动调用 Circle 类对应的无参构造函数，将对象成员 c0、c1、c2 的半径初始化为 0。

■ TriCircle obj2( obj );

该定义语句用已经存在的对象 obj 来初始化新建对象 obj2。根据实参—形参匹配原则，执行该语句将自动调用拷贝构造函数(3)来初始化 obj2。初始化列表将形参 rObj 所引用的实参 obj 中的 3 个对象成员 c0、c1、c2 接力传递给 Circle 类拷贝构造函数，分别复制给 obj2 中对应的成员。初始化后，新建对象 obj2 中 3 个对象成员 c0、c1、c2 的半径也是 5、2、3，与原有对象 obj 完全一样。

#### 2. 组合类的析构函数

当对象生存期结束时，计算机销毁对象，释放其内存空间，这个过程就是对象的析构。销毁对象时计算机会自动调用其所属类的析构函数。类的析构函数只有一个，其功能是清理内存，例如释放构造对象时所申请的额外内存。

组合类对象中数据成员的析构顺序是：先执行组合类析构函数的函数体，清理非对象成员；再自动调用对象成员所属类的析构函数，清理对象成员。简单地说，组合类对象的析构顺序与构造顺序相反，即先析构非对象成员，再析构对象成员。

### 8.2.4 类的聚合

例 8-2 定义的一个描述图 8-4(a)中几何图形的类 TriCircle，也可以用例 8-3 所定义的类 pTriCircle 来描述该图形。类 pTriCircle 中声明了 3 个数据成员 p0、p1、p2，它们是 Circle 类的对象指针。C++语言将数据成员中包含对象成员的类称为组合类，而将数据成员中包含对象指针的类称为聚合类。聚合类是一种特殊形式的组合类。

**例 8-3 聚合类 pTriCircle 的定义代码**

类声明头文件：pTriCircle.h

```
#include "Circle.h"   // 声明类 Circle

class pTriCircle   // 声明部分，即声明成员
{
public:
    // 声明 3 个公有的 Circle 类对象指针
    Circle *p0, *p1, *p2;

    // 以下为 pTriCircle 类相关的新内容
    double TArea( );   // 求总面积：公有函数成员
    double TLen( );   // 求总周长：公有函数成员
};
```

类实现程序文件：pTriCircle.cpp

```
#include "pTriCircle.h"   // 声明类 pTriCircle

// pTriCircle 类：实现部分
double pTriCircle :: TArea( )   // 求总面积
{
    double totalArea;
    totalArea = p0->CArea( ) +
                p1->CArea( ) + p2->CArea( );
    return   totalArea;
}
double pTriCircle :: TLen( )   // 求总周长
{
    double totalLen;
```

| | |
|---|---|
| | totalLen = p0->CLen ( ) +<br>p1->CLen( ) + p2->CLen( );<br>return totalLen;<br>} |

下面的主函数演示了聚合类 pTriCircle 的使用方法，其程序功能是计算图 8-4(a)所示几何图形的总面积和总周长。

```
#include <iostream>
using namespace std;
#include "pTriCircle.h"                          // 类 pTriCircle 的声明头文件
int main( )
{
      Circle c0, c1, c2;                         // 先定义 3 个类 Circle 的对象 c0、c1、c2
      c0.Input( );   c1.Input( );   c2.Input( );  // 分别输入 3 个圆的半径
      pTriCircle   obj1;                         // 定义一个聚合类 pTriCircle 的对象 obj1
      // 将 obj1 中 3 个对象指针分别指向前面创建的 Circle 类对象 c0、c1、c2
      obj1.p0 = &c0;   obj1.p1 = &c1;   obj1.p2 = &c2;
      // 调用 obj1 中的函数成员 TArea 和 TLen，计算并显示总面积和总周长
      cout << obj1.TArea( ) << ", " << obj1.TLen( ) << endl;

      pTriCircle   obj2;                         // 再定义一个聚合类 pTriCircle 的对象 obj2
      // 将 obj2 中 3 个对象指针也分别指向 Circle 类对象 c0、c1、c2
      obj2.p0 = &c0;   obj2.p1 = &c1;   obj2.p2 = &c2;
      // 调用 obj2 中的函数成员 TArea 和 TLen，计算并显示总面积和总周长
      cout << obj2.TArea( ) << ", " << obj2.TLen( ) << endl;
}
```

聚合类与组合类的区别有两点：一是聚合类的对象成员是独立创建的，聚合类对象只包含指向对象成员的指针；二是聚合类对象可以共用对象成员。例如上例中，3 个 Circle 类对象 c0、c1、c2 都是独立创建的对象，聚合类对象 obj1 和 obj2 中的 3 个对象指针 p0、p1、p2 都分别指向了这 3 个 Circle 类对象。obj1 和 obj2 共用对象成员（如图 8-7 所示）。聚合类对象可以共用对象成员，在内存中只需保存一份对象成员的数据，这样可以节省内存。

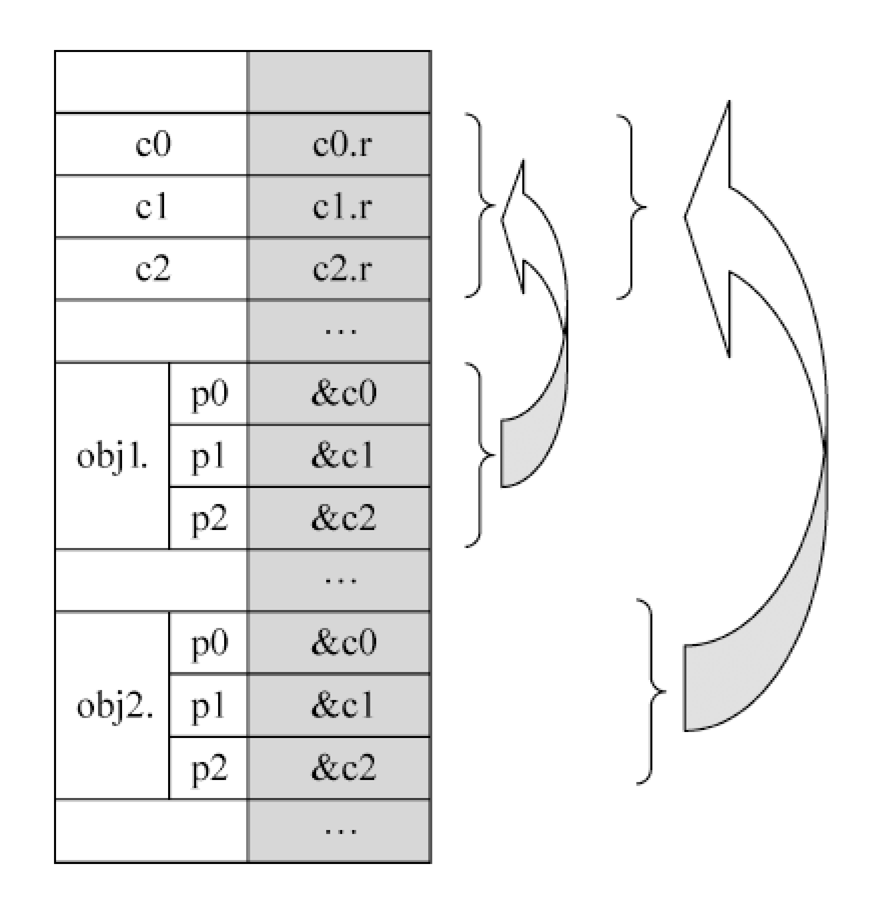



图 8-7　聚合类对象 obj1 和 obj2 共用对象成员的示意图

### 8.2.5 前向引用声明

C++语言中的类需“先定义，再使用”，或“先声明，再使用”。某些特殊情况会出现两个类互相使用对方，即相互依赖。例如下面的两个类 A 和 B：

```
class A                    // 类 A 是个聚合类，其中定义了一个类 B 的对象指针 pbObj
{
public:
    B  *pbObj;             // 类 A 使用了类 B
    // ……其他代码省略
} ;
class B                    // 类 B 也是个聚合类，其中定义了一个类 A 的对象指针 paObj
{
public:
    A  *paObj;             // 类 B 也使用了类 A
    // ……其他代码省略
} ;
```

在这个例子中，无论将哪个类放在前面，编译时都会出现错误。例如上述代码在编译时将提示“标识符 B 未定义”。为此，C++语言引入了**前向引用声明**。在上述代码类 A 的定义之前加上类 B 的前向引用声明，这样编译就可以通过了。

```
class B;                   // 类 B 的前向引用声明
class A                    // 类 A 是个聚合类，其中定义了一个类 B 的对象指针 pbObj
{  // ……代码省略  } ;
class B                    // 类 B 也是个聚合类，其中定义了一个类 A 的对象指针 paObj
{  // ……代码省略  } ;
```

虽然通过前向引用声明解决了上述类 A 和 B 之间的相互依赖问题，但这仅仅是一个特例。例如，下面组合类的例子就不能用前向引用声明来解决问题了。

```
class B;                   // 类 B 的前向引用声明
class A                    // 类 A 中定义了类 B 的对象成员，即类 A 是个组合类
{
public:
    B  bObj;               // 错误：在掌握类 B 的完整定义之前，编译器无法编译该语句
} ;
class B                    // 类 B 中定义了类 A 的对象成员，即类 B 也是个组合类
{
public:
    A  aObj;
} ;
```

程序员在设计类的时候应尽量避免类之间的相互依赖，否则很容易陷入这种语法陷阱。所谓**语法陷阱**，就是虽然 C++语言提供了某种语法形式，但应用时很容易出错。再看下面的两个例子：

```
int x;  cout << (x=3+4, x*5), x+6;          // 执行该语句后的显示结果为 35，你答对了吗？
for (int x=0, y=5; – –y > 0; x++, y– –);    // 执行该语句后 x、y 的值分别为 2 和 0，你答对了吗？
```

上述语句的语法都是正确的，但非常晦涩难懂。某些 C++语言考试可能会使用这种题目，很考验你的语法基本功。但在实际编程中最好不要使用这种句型，否则既容易出错，又会给今后的代码阅读或维护修改带来麻烦。

本节最后再对组合类做一个简单总结：

（1）**代码重用**。组合类是一种有效的重用类代码形式。程序员在设计新类的时候应先去了解有哪些可以重用的类。这些类可以是自己以前编写的，可以是集成开发环境 IDE 提供的，也可以是从市场上购买来的。根据功能选择自己需要的类，然后用组合的方法定义新类。

（2）**多级组合**。用零件类定义组合类，组合类可继续作为零件类去定义更大的组合类，这就是类的多级组合。多级组合是一种“自底向上”的程序设计方法。类越往上组合，其功能就越有针对性，应用面也就越窄。多级组合过程中，每一级组合类都会根据自己的功能需要设定对象成员的访问权限，即多层封装。有多少级组合，就有多少层封装。

## 本节习题

1. 下列关于组合类的描述中，正确的是（　　）。
   A. 数据成员中包含类类型的对象成员，这样的类被称为组合类
   B. 函数成员访问了类类型对象的数据成员，这样的类被称为组合类
   C. 函数成员调用了类类型对象的函数成员，这样的类被称为组合类
   D. 组合类数据成员中不能包含非对象成员，即用基本数据类型定义的变量
2. 下列关于组合类对象成员的描述中，错误的是（　　）。
   A. 所谓对象成员，就是用类定义的对象
   B. 对象成员还包含下级成员
   C. 组合类设定对象成员的访问权限是对其进行二次封装
   D. 组合类中的函数成员访问对象成员的下级成员不受权限控制
3. 下列关于组合类对象的描述中，错误的是（　　）。
   A. 组合类所定义的对象中包含对象成员
   B. 访问组合类对象的下级成员是多级访问
   C. 访问组合类对象的下级成员需受多级权限的控制
   D. 不能用对象指针访问组合类对象及其下级成员
4. 定义类 A 和组合类 B：

```
class A
{
private:    int x;
public:    int y;
};
class B
{
public:    A t;    int s;
};
```

使用组合类 B 定义对象“B obj;”下列语句中正确的是（　　）。

A. obj.x = 5;　obj.y = 5;　obj.s = 5;

B. obj.t.x = 5;　obj.t.y = 5;　obj.s = 5;

C. B *p = &obj;　p.t.x = 5;　p.t.y = 5;　p.s = 5;

D. B *p = &obj;　p->t.y = 5;　p->s = 5;

5. 下列关于组合类构造函数和析构函数的描述中，错误的是（　　）。

A. 组合类构造函数通过初始化列表调用对象成员的构造函数，实现对象成员的初始化

B. 组合类析构函数自动调用对象成员的析构函数，实现对象成员销毁之前的清理工作

C. 创建组合类对象时先调用对象成员的构造函数，再执行组合类构造函数的函数体

D. 销毁组合类对象时先调用对象成员的析构函数，再执行组合类析构函数的函数体

## 8.3 类的继承与派生

遗传与变异是生物进化的基础，是继承祖先优良品质，同时不断进化以适应新的生存环境的重要机制。面向对象程序设计借鉴了这种机制，为类代码提供一种新的，也是最为重要的一种代码重用形式，这就是类的**继承**（inheritance）与**派生**（derivation）。

程序员针对新的程序设计问题可能需要设计新的类。设计新类时可以继承已有的类，这个已有的类被称为**基类**（base class）或**父类**（father class）。继承的目的是重用已有类的代码，从而提高新类的开发效率。

如果仅仅是单纯继承基类，那就只是简单的克隆，在程序设计中没有实际意义。在继承基类的同时添加新功能，或者对从基类继承来的功能进行升级改造，这被称为对基类进行**派生**。派生的目的是为了解决新问题。通过继承与派生所得到的新类被称为**派生类**（derived class）或**子类**（child class）。

继承与派生的**编程原理**是：程序员在定义新类的时候，首先继承基类的数据成员和函数成员；在此基础上进行派生，为派生类添加新成员，或对从基类继承来的成员进行重新定义或修改其访问权限。在继承与派生的过程中，继承实现了基类代码的**重用**，派生则实现了基类代码的**进化**。派生类是对基类的继承和发展，使用派生类可以解决新的程序设计问题。

按照来源的不同，派生类中的成员可分为两种：一是从基类继承来的成员，称为**基类成员**，二是定义时新增的成员，称为**新增成员**。

## 8.3.1　派生类的定义

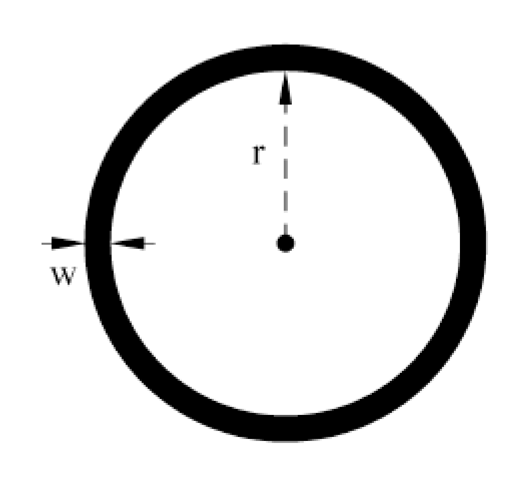
C++语法：定义派生类

```
class 派生类名 : 继承方式 1 基类 1, 继承方式 2 基类 2……  // 派生类声明部分
{
public :
    新增公有成员
protected :
    新增保护成员
private :
    新增私有成员
} ;
各新增函数成员的完整定义  // 派生类实现部分
```

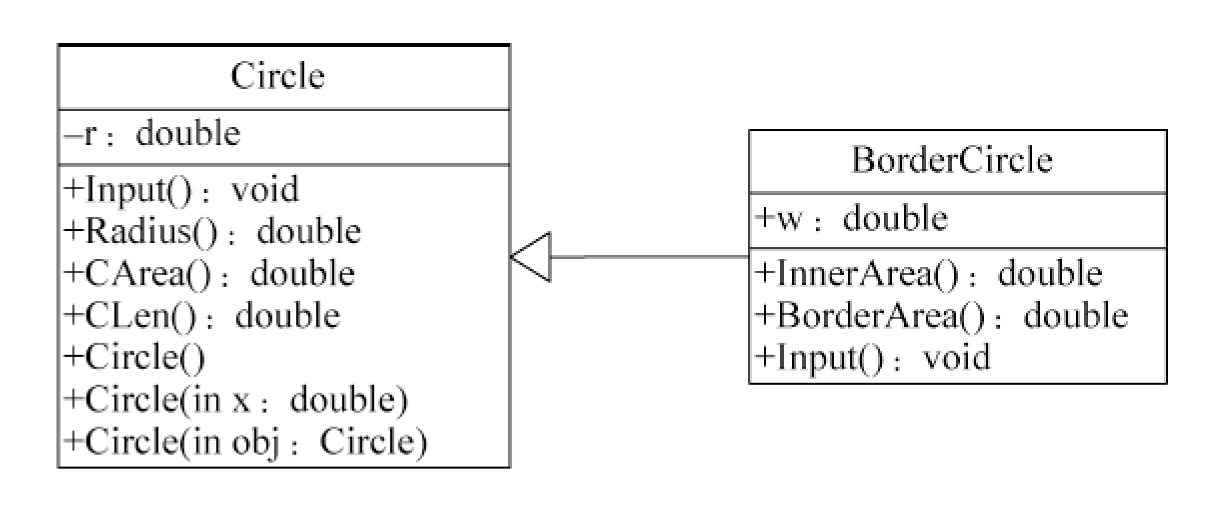
语法说明：

- 定义派生类时，在派生类名的后面添加**继承列表**，在声明部分的大括号里声明**新增成员**，在实现部分编写各新增函数成员的完整定义代码。新增成员的目地是扩充或修改基类的功能。
- **继承列表**指定派生类从哪些基类继承。派生类可以只从一个基类继承（**单继承**），也可以从多个基类继承（**多继承**）。每个基类以“**继承方式 基类名**”的形式声明，多个基类之间用“,”隔开。
- 派生类将继承基类中**除构造函数、析构函数之外**的所有数据成员和函数成员。基类的构造函数和析构函数不能被继承，派生类需重新编写自己的构造函数和析构函数。
- 继承后，派生类会对其基类成员按照**继承方式**进行**再次封装**。继承方式有三种：public（公有继承）、protected（保护继承）和 private（私有继承）。
  - public（**公有继承**）：派生类对其基类成员不做任何封装，它们在派生类中的访问权限与原来在基类中的权限相同。
  - private（**私有继承**）：派生类对其基类成员做全封装，它们在派生类中的访问权限统统被改为 private（私有权限），无论它们原来在基类中的权限是什么。使用私有继承，实际上是派生类要将其基类成员全部隐藏起来。
  - protected（**保护继承**）：派生类对其基类成员做半封装。基类中的 public 成员被继承到派生类后，其访问权限被降格成 protected（保护权限）。基类中的 protected、private 成员被继承到派生类后，其访问权限保持不变。
- 在**类声明部分**的大括号中声明新增的数据成员、函数成员，并指定各新增成员的访问权限。在**类实现部分**编写各新增函数成员的完整定义代码。

假设我们要定义一个描述图 8-8(a)中带边框圆形的类 BorderCircle。带边框圆形有 2 个属性，即半径 r 和边框的宽度 w，对带边框圆形的处理可能包括计算其面积、周长，以及内圆面积和边框面积。




(a) 带边框的圆形




(b) 类之间的继承关系

图 8-8　派生类举例

定义 BorderCircle 类可以从零开始编写，也可以基于例 8-1 的 Circle 类来编写派生类。通过继承 Circle 类，派生类 BorderCircle 就不需要再重复定义数据成员半径 r、输入半径函数 Input、求圆形面积函数 CArea 和求圆形周长函数 CLen，只需要补充与边框相关的新内容，例如新增边框宽度 w、求内圆面积和边框面积的函数 InnerArea、BorderArea。新增的 w、InnerArea 和 BorderArea 就是对基类 Circle 的扩充。

Circle 类中的输入函数 Input 只能输入半径，为此 BorderCircle 类重新定义了一个 Input 函数，这相当于是修改了原 Input 函数。新的 Input 函数能同时输入半径和边框宽度。

图 8-8(b)用 UML 统一建模语言描述了这两个类之间的继承关系。通过继承 Circle 类来定义派生类 BorderCircle，其定义代码如例 8-4 所示。

**例 8-4　派生类 BorderCircle 的定义代码**

类声明头文件：BorderCircle.h

```
#include "Circle.h"    // 声明基类 Circle

// 以下为派生类 BorderCircle 的声明部分
class BorderCircle : public Circle    // 公有继承 Circle 类
{
public:
    double w;    // 新增数据成员：边框宽度

    // 以下为新增的函数成员
    double InnerArea( );    // 求内圆面积
    double BorderArea( );    // 求边框面积
    void Input( );    // 输入半径和边框宽度
};
```

类实现程序文件：BorderCircle.cpp

```
#include <iostream>
using namespace std;
#include "BorderCircle.h"

// BorderCircle 类：实现部分
// 编写新增函数成员的完整定义代码
double BorderCircle :: InnerArea( )
{
    double x = Radius( );    // 读取半径
    return   ( 3.14 * (x-w) * (x-w) );
}
double BorderCircle :: BorderArea( )
{   return   ( CArea( ) - InnerArea( ) );   }
void BorderCircle :: Input( )
{
    Circle :: Input( );    // 输入半径
    cin >> w;    // 输入边框宽度
}
```

例 8-4 的程序说明如下。

**1）派生类 BorderCircle 中的基类成员**

例 8-4 所定义的派生类 BorderCircle 继承了基类 Circle 中除构造函数、析构函数之外的所有数据成员和函数成员，它们分别是：

- 数据成员：半径 r。
- 函数成员：输入半径函数 Input、读取半径函数 Radius、求面积函数 CArea 和求周长函数 CLen。

以上 5 个成员被称为是派生类 BorderCircle 的基类成员，因为它们是从基类 Circle 中继承来的。派生类 BorderCircle 中还定义了 4 个新增成员，它们分别是：

- 数据成员：边框宽度 w。
- 函数成员：求内圆面积函数 InnerArea、求边框面积函数 BorderArea 和一个重写的 Input 函数。

派生类 BorderCircle 总共包含 9 个成员，其中 5 个是基类成员，4 个是新增成员。

**2）在派生类中访问基类成员**

派生类中的新增成员可以访问继承来的基类成员，但要受到它们原来在基类中访问权限的限制。例如，派生类新增函数成员 InnerArea 在计算内圆面积时需要用到半径。但半径 r 是基类成员，它在基类 Circle 中是私有的，不能直接访问。因此 InnerArea 要调用公有的基类成员 Radius 来间接读取半径。

**3）派生类中新增成员对基类成员的同名覆盖**

另外需要注意的是，新增函数成员可以与基类函数成员重名，但它们不是重载函数。例如派生类 BorderCircle 继承了基类 Circle 中的输入半径函数 Input，然后又新增了一个函数成员 Input，这两个函数重名了。

在派生类中定义与基类成员重名的新增成员，新增成员将**覆盖**（override）基类成员。通过成员名访问时，所访问到的将是新增成员，这就是新增成员对基类成员的**同名覆盖**。同名覆盖后，被覆盖的基类成员仍然存在，只是被隐藏了。可以访问被覆盖的基类成员，其访问形式是：

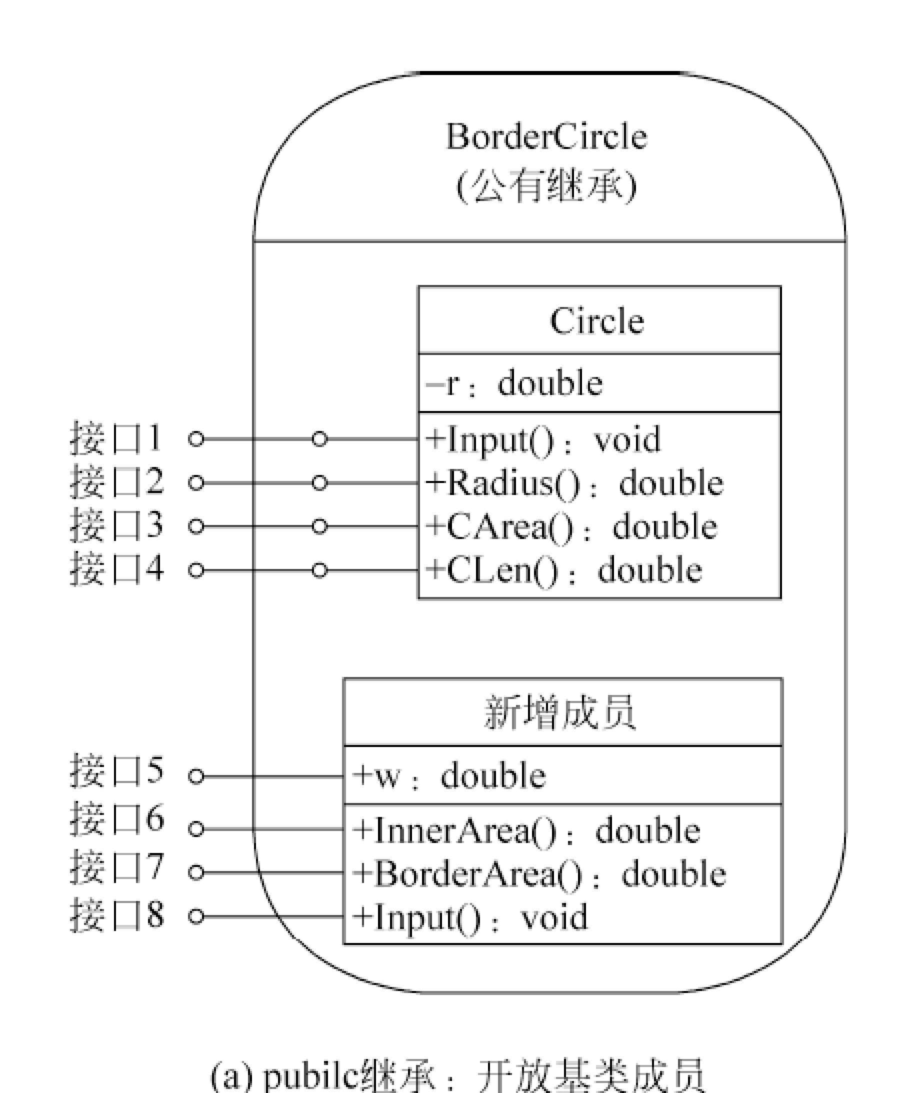
**基类名 :: 基类成员名**

同名覆盖的目的是为了修改或升级基类成员的功能。例如，基类 Circle 中的 Input 函数只能输入半径，因此派生类 BorderCircle 重新定义一个 Input 函数，它能够同时输入半径和边框宽度。输入半径时，新增的 Input 函数不能直接访问私有的基类成员半径 r，必须调用被覆盖的公有基类成员 Input 才能实现输入半径的功能，其调用形式如下：

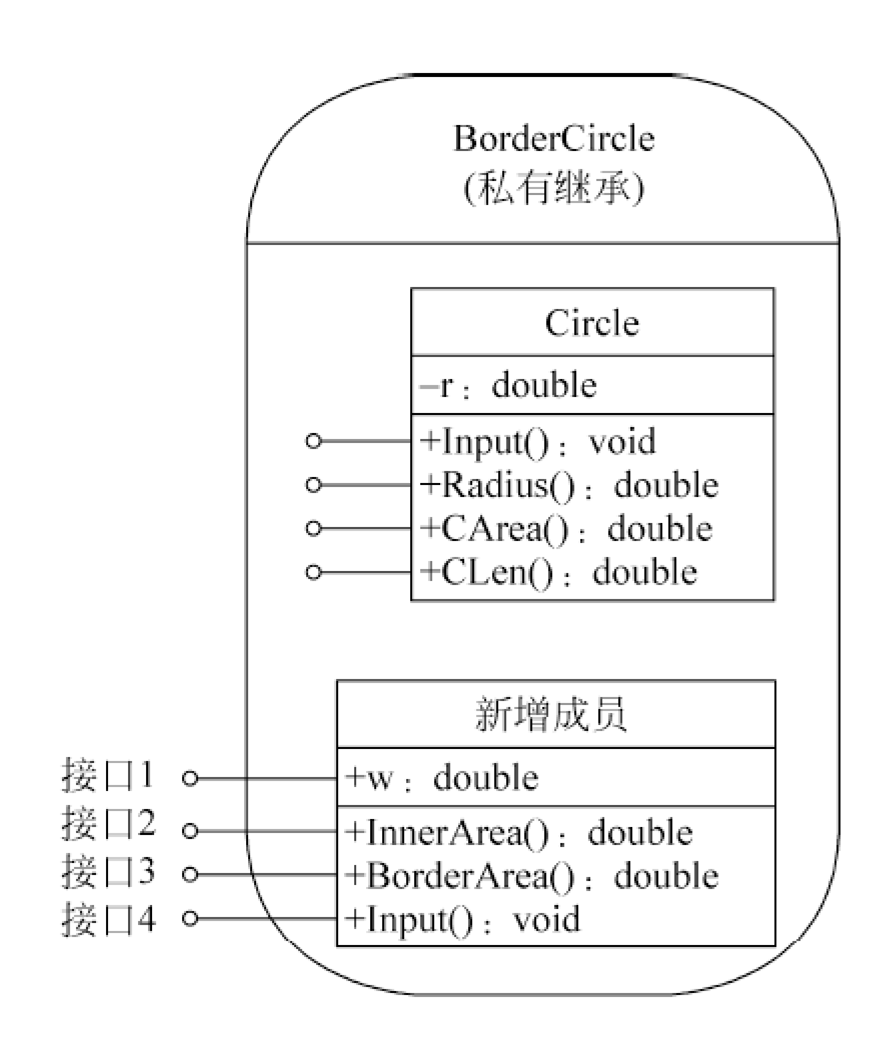
```
Circle :: Input( );   // 调用公有基类成员 Input，为私有基类成员半径 r 输入数据
```

**4）派生类对基类成员的二次封装**

派生类通过继承方式对从基类继承来的成员进行二次封装。公有继承 public 就是在派生类中继续开放基类成员中的公有成员（相当于没有封装），如图 8-9(a)所示。而私有继承 private 则是将所有基类成员统统隐藏起来（即全封装），包括其中的公有成员，如图 8-9(b)所示。此时，这些基类成员只能在派生类的内部才能访问，在类的外部不能访问。



(a) pubilc继承：开放基类成员



(b) private继承：隐藏基类成员

图 8-9　派生类对基类成员的二次封装

## 8.3.2　派生类对象的定义与访问

与任何普通的类一样，可以使用派生类来定义对象。例如，定义一个派生类 BorderCircle 的对象 obj：

```
BorderCircle    obj;   //  定义一个派生类 BorderCircle 的对象 obj
```

计算机执行该对象定义语句时将为对象 obj 分配内存空间（如图 8-10 所示）。一个派生类对象所占用的内存空间等于类中全部数据成员（含基类成员）所需内存空间的总和。

### 1. 派生类对象中基类成员的访问

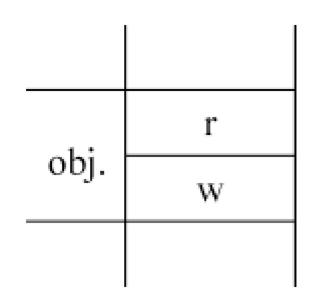


图 8-10　派生类对象 obj 的内存示意图

按照 BorderCircle 类的定义，对象 obj 将包含如下 9 个成员：

- **5 个基类成员**：1 个数据成员半径 r，4 个函数成员 Input、Radius、CArea 和 CLen。派生类 BorderCircle 公有继承基类 Circle，没有对基类成员进行二次封装，因此这 5 个基类成员在派生类中的访问权限与它们原来在基类中的权限相同，即数据成员半径 r 仍是私有的，而另外 4 个函数成员仍是公有的。
- **4 个新增成员**：1 个数据成员边框宽度 w，以及 3 个函数成员 InnerArea、BorderArea 和 Input。BorderCircle 类在定义时将这 4 个新增成员都设定为公有权限。

派生类对象中的成员，无论是基类成员还是新增成员，只要是公有权限就都可以访问，是非公有权限（私有权限、保护权限）就都不能访问。例如，可以访问派生类对象 obj 中的公有成员，其访问形式分别是：

- 访问基类成员：obj.Circle::Input( )、obj.Radius( )、obj.CArea( )、obj.CLen( )

■ 访问新增成员：obj.w、obj.InnerArea( )、obj.BorderArea( )、obj.Input( )

可以看出，访问派生类中的成员，无论是基类成员还是新增成员，它们的访问形式都是一样的。只是在同名覆盖的情况下，访问被覆盖的基类成员时需在成员名前加前缀“基类名 ::”，显式指明基类成员，否则默认的都是新增成员。

下面的主函数演示了如何利用派生类对象来计算图 8-8(a)所示带边框圆形的面积、周长，以及内圆面积和边框面积。

```
#include <iostream>
using namespace std;
#include "BorderCircle.h"            // 类 BorderCircle 的声明头文件
int main( )
{
    BorderCircle   obj;              // 定义一个派生类 BorderCircle 的对象 obj

    // 调用派生类对象 obj 中的新增函数成员 Input，输入半径和边框宽度
    obj.Input( );                    // 调用的是新增成员 Input，重名的基类成员 Input 被覆盖了

    // 调用 obj 中的基类函数成员 CArea 和 CLen，计算并显示圆的面积和周长
    cout << obj.CArea( ) << ", " << obj.CLen( ) << endl;

    // 调用 obj 中的新增成员 InnerArea 和 BorderArea，计算并显示内圆和边框的面积
    cout << obj.InnerArea( ) << ", " << obj.BorderArea( ) << endl;
}
```

### 2. 如何设计派生类的继承方式

在派生类的定义和使用过程中有三个程序员角色，程序员甲是编写基类的程序员，程序员乙是编写派生类的程序员，程序员丙则是使用派生类定义对象的程序员。

**程序员乙**定义派生类，其中包含从基类继承来的成员。定义派生类时，程序员乙应根据功能要求决定如何将基类成员开放给程序员丙。设定公有继承就是将基类成员原样开放给程序员丙，设定私有继承就是将基类成员全部隐藏起来，这就是继承时基类成员的二次封装。例如在定义派生类 BorderCircle 时，程序员乙根据功能要求决定将基类成员原样开放给程序员丙，即将 Circle 类的继承方式设为公有继承 public。

**程序员丙**使用派生类定义对象，然后访问派生类对象的下级成员。程序员丙能够访问哪些下级成员将取决于访问权限，只有公有权限的成员才能访问，否则就不能访问。派生类对象包含基类成员和新增成员。其中新增成员的访问权限是程序员乙在定义派生类时直接设定的，比较容易判断，而要判断基类成员的访问权限则稍微有点复杂。

派生类中基类成员的访问权限由两个因素决定：一是派生类继承基类时的继承方式，二是基类成员原来在基类中的访问权限。换句话说，派生类中基类成员的访问权限是由程序员乙和程序员甲两个人共同决定的。

（1）**程序员乙**：如果程序员乙在定义派生类时选择私有继承，将基类成员全部隐藏起来，则程序员丙使用派生类所定义对象中的基类成员统统不可以访问。如果程序员乙定义派生类时选择公有继承，继续开放基类成员，则这些基类成员中哪些能访问、哪些不能访

问将取决于程序员甲。

（2）**程序员甲**：如果基类成员原来在基类中的权限是公有权限，则程序员丙可以访问，否则不能访问。换句话说，派生类公有继承基类，不对基类成员进行二次封装，则原本开放的基类成员可以访问，原本未开放的基类成员则不能访问。

总结一下，程序员丙在访问派生类对象中的基类成员时，只有公有继承下的公有成员才可以访问，否则就不能访问。

#### 3. 多级派生和多种派生

派生类也可以任意多级。用基类定义派生类，派生类可以继续作为基类去定义更下级的派生类，这就是**多级派生**。多级派生过程中，每一级派生类都会根据自己的功能需要设定继承方式，这相当于对所继承的基类成员进行再次封装，即多层封装。

多个不同的派生类可以继承同一个基类。换句话说，同一个基类可以派生出多个不同的派生类，这就是**多种派生**。同一基类可经过多种派生和多级派生，派生出很多个不同的派生类，这个基类和它的派生类一起共同组成了一个类的家族，即**类族**。类族中的类具有共同的特性，因为它们具有共同的祖先，即都继承了同一个基类。

### 8.3.3 保护权限与保护继承

#### 1. 类成员的保护权限

类成员的访问权限有三种。

（1）**公有权限**（public）。被赋予公有权限的类成员是开放的，称为公有成员。

（2）**私有权限**（private）。被赋予私有权限的类成员将被隐藏，称为私有成员。

（3）**保护权限**（protected）。被赋予保护权限的类成员是半开放的，称为保护成员。

其中保护权限的半开放是什么意思？保护权限对谁开放，对谁不开放呢？请先看一个例子：程序员甲、乙、丙 3 人，甲编写了一个类 A。乙使用类 A 定义对象，编写应用程序。而丙则是使用类 A 去另外定义一个新的派生类 B。具体代码如例 8-5 所示。

**例 8-5　关于保护权限的 C++演示程序**

**程序员甲：**定义类 A（类声明头文件 A.h）

```
class A                                   // 定义类 A
{
public:
    A( int p1 = 0, int p2 = 0, int p3 = 0 )  // 构造函数
    {   x = p1;   y = p2;   z = p3;   }
    int   x;                              // 公有权限
private:   int   y;                       // 私有权限
protected:   int   z;                     // 保护权限
} ;
// 类 A 没有函数成员，所以不需要类实现部分
```

**程序员乙：使用类 A 定义对象（1.cpp）**

```
#include <iostream>
using namespace std;
#include "A.h"                    // 声明类 A

int main( )
{
    A   obj( 10, 20, 30 );
    cout << obj.x << endl;   // 正确：public
    cout << obj.y << endl;   // 错误：private
    cout << obj.z << endl;   // 错误：protected
    return 0;
}
```

**程序员丙：使用类 A 定义派生类 B（B.h）**

```
#include <iostream>
using namespace std;
#include "A.h"        // 声明基类 A

class B : public A
{
public:
    void funB( )   // 新增成员访问基类成员
    {
        cout << x << endl;   // 正确：public
        cout << y << endl;   // 错误：private
        cout << z << endl;   // 正确：protected
    }
} ;
```

例 8-5 中，**程序员甲**将类 A 中 3 个数据成员 x、y、z 分别设定为公有权限、私有权限和保护权限。我们通过这个例子来具体分析一下程序员甲在类 A 中设定保护权限将对程序员乙和丙产生什么不同的影响。

- **程序员乙**。乙使用类 A 定义对象。在主函数 main 中定义 A 类对象 obj，并访问对象 obj 的 3 个成员 x、y、z。通过对象访问其成员，只能访问对象中的公有成员（例如 x），不能访问私有成员（例如 y）和保护成员（例如 z）。对程序员乙来说，在类外的主函数 main 中访问对象的成员，保护成员和私有成员一样，都不能访问。
- **程序员丙**。丙使用类 A 去定义派生类 B，同时新增一个函数成员 funB，并在其中访问 3 个基类成员 x、y、z。派生类中新增函数成员访问基类成员，可以访问原来在基类中公有的成员（例如 x）和保护成员（例如 z），不能访问私有成员（例如 y）。对程序员丙来说，在派生类新增函数成员中访问基类成员，基类中的保护成员和公有成员一样，都可以访问。

站在程序员甲的角度看，乙编写的主函数 main 和丙编写的派生类新增函数成员 funB 都是类 A 的外部函数。公有权限或私有权限对类外所有函数都一视同仁，具有相同的约束力。

保护权限则不同，保护成员对派生类中的新增函数成员来说是开放的，但对类外的所有其他函数来说却是隐藏的，这就是所谓的半开放。例如类 A 中的保护成员 z，丙编写的派生类新增函数成员 funB 可以访问从类 A 中继承来的保护成员 z，但乙编写的主函数 main 却不能访问 A 类对象 obj 的保护成员 z。类的保护权限是向其派生类定向开放的一种权限。

### 2. 派生类的保护继承

派生类通过继承方式对从基类继承来的成员进行二次封装。公有继承 public 就是在派生类中将基类成员原样开放（相当于没有封装）。而私有继承 private 则是将所有基类成员统统隐藏起来（即全封装），包括其中的公有成员。

保护继承 protected 则是一种半封装，这个半封装又是什么意思呢？请继续看下面的例子：程序员甲、乙、丙 3 人，甲通过继承例 8-5 中的类 A 来定义派生类 B，继承时使用保

护继承。乙使用派生类 B 定义对象，编写应用程序。丙则是使用派生类 B 再去定义一个新的派生类 C，即多级派生。具体代码如例 8-6 所示。

**例 8-6　关于保护继承的 C++演示程序**

**程序员甲：使用例 8-5 中的类 A 定义派生类 B（类声明头文件 B.h）**

```
#include <iostream>
using namespace std;
#include "A.h"                    // 声明基类 A

class B : protected   A           // 保护继承
{
public:
    void funB( )                  // 新增成员访问基类成员
    {
        cout << x << endl;        // 正确：public
        cout << y << endl;        // 错误：private
        cout << z << endl;        // 正确：protected
    }
    B(int p1, int p2, int p3) : A(p1, p2, p3) { }   // 派生类构造函数
} ;
```

**程序员乙：使用类 B 定义对象（1.cpp）**

```
#include <iostream>
using namespace std;
#include "B.h"   // 声明类 B

int main( )
{
    B   obj( 10, 20, 30 );
    cout << obj.x << endl; //  错误：protected
    cout << obj.y << endl; //  错误：private
    cout << obj.z << endl; //  错误：protected
    return 0;
}
```

**程序员丙：使用类 B 定义派生类 C（C.h）**

```
#include <iostream>
using namespace std;
#include "B.h"   // 声明基类 B

class C : protected   B   // 两级继承基类 A
{
public:
    void funC( )   // 新增成员访问基类成员
    {
        cout << x << endl;   // 正确：protected
        cout << y << endl;   // 错误：private
        cout << z << endl;   // 正确：protected
    }
} ;
```

我们通过这个例子来具体分析一下，**程序员甲**在定义派生类 B 时使用保护继承将对程序员乙和丙产生什么不同的影响。保护继承时，基类中的 public 成员被继承到派生类后，其访问权限被降格成 protected（保护权限）；基类中的 protected、private 成员被继承到派生类后，其访问权限保持不变。

例 8-5 中类 A 的 3 个数据成员 x、y、z 的访问权限分别是：

```
public:      int   x;          // 公有权限
private:     int   y;          // 私有权限
protected:   int   z;          // 保护权限
```

派生类 B 继承类 A 的这 3 个成员，保护继承后它们在派生类 B 中的访问权限分别为：

```
protected:  int  x;        // 公有权限被降格成保护权限
private:    int  y;        // 私有权限保持不变
protected:  int  z;        // 保护权限保持不变
```

- **程序员乙**。乙使用派生类 B 定义对象，在主函数 main 中定义 B 类对象 obj，但不能访问对象 obj 中的 3 个基类成员 x、y、z。因为通过对象访问其成员，只能访问对象中的公有成员，私有成员（例如 y）和保护成员（例如 x 和 z）不能访问。对程序员乙来说，派生类保护继承基类，这意味着派生类对象中的所有基类成员都被隐藏起来了，不能访问。
- **程序员丙**。丙使用派生类 B 继续去定义一个新的派生类 C，同时新增一个函数成员 funC，并在其中访问从类 B 中继承来的 3 个基类成员 x、y、z。派生类中新增函数成员访问基类成员，可以访问基类中的公有成员和保护成员（例如 x 和 z），不能访问私有成员（例如 y）。对程序员丙来说，派生类保护继承基类，这意味着派生类将从上级基类中继承来的公有和保护成员继续向其下级派生类开放。

站在程序员甲的角度看，乙编写的主函数 main 和丙编写的下级派生类新增函数成员 funC 都是派生类 B 的外部函数。派生类通过继承方式对从基类继承来的成员进行二次封装。公有继承或私有继承对派生类外部的所有函数都一视同仁，要么是都不封装，要么是都封装。

保护继承则不同。派生类保护继承基类，对下级派生类的新增函数成员来说，派生类中的基类成员没被封装；但对派生类外部的所有其他函数来说，这些基类成员却被封装起来了，这就是所谓的半封装。例如类 A 中的公有成员 x 和保护成员 z，派生类 B 保护继承类 A，对丙编写的下级派生类新增函数成员 funC 来说它们没被封装，可以访问；但对乙编写的主函数 main 来说它们被封装了，不能访问。派生类的保护继承是向其下级派生类定向开放基类成员的一种半封装形式。

## 8.3.4　派生类对象的构造与析构

按照来源的不同，派生类中的成员可分为两种：一是从基类继承来的成员，称为派生类中的基类成员；二是定义时新增的成员，称为派生类中的新增成员。初始化基类成员比较麻烦，因为基类成员可能是私有的，不能直接访问赋值。

### 1. 派生类的构造函数

构造函数通过形参传递初始值，实现对新建对象数据成员的初始化。派生类构造函数不能直接初始化类中的基类成员，因为它们在基类中可能是私有的，不能访问赋值。要想初始化这些基类成员，必须通过基类的构造函数才能完成。调用基类构造函数，其语法形式是在派生类构造函数的函数头后面添加**初始化列表**（黑体部分）：

```
派生类构造函数名( 形参列表 ): 基类名 1( 形参 1 ), 基类名 2( 形参 2 ) …
{
…  // 在函数体中初始化新增成员
}
```

其中，形参 1、形参 2 等是从形参列表中提取出来的，并在初始化列表中进行二次接力传递。派生类对象中各数据成员的初始化顺序是：先调用基类构造函数，初始化基类成员；再执行派生类构造函数的函数体，初始化新增成员。如果派生类继承了多个基类，那么各基类的初始化顺序由其在派生类继承列表中的声明顺序决定，声明在前的基类成员先初始化。

例如，为派生类 BorderCircle 添加如下 3 个重载构造函数：

（1）**有参构造函数**。为了初始化半径 r 和边框宽度 w，需传递 2 个初始值。

```
// 通过初始化列表初始化基类成员：半径 r
BorderCircle :: BorderCircle( double p1, double p2 ) : Circle(p1)
{   w = p2;   }                  // 在函数体中初始化新增成员：边框宽度 w
```

（2）无参构造函数。

```
BorderCircle :: BorderCircle( )   {   w = 0;   }
```

（3）**拷贝构造函数**。接收一个已经存在的本类对象引用，用该对象来初始化新建对象。

```
BorderCircle :: BorderCircle( BorderCircle &rObj ) : Circle( rObj )
{   w = rObj.w;   }           // 在函数体中初始化新增成员：边框宽度 w
```

上述 3 个构造函数为 BorderCircle 类对象提供了 3 种不同的初始化方式，例如：

■ BorderCircle　obj( 5, 2 );

该定义语句给出 2 个实参（即初始值）。根据实参—形参匹配原则，执行该语句将自动调用有参构造函数(1)来初始化新建对象 obj。初始化列表将形参 p1 所接收到的实参数据接力传递给 Circle 类对应的有参构造函数，将基类成员半径 r 初始化为 5。然后构造函数的函数体用形参 p2 所接收到的实参数据为新增成员边框宽度 w 赋值，将其初始化为 2。

■ BorderCircle　obj1;

该定义语句没有给实参。根据实参—形参匹配原则，执行该语句将自动调用无参构造函数(2)来初始化新建对象 obj1。虽然无参构造函数没写初始化列表，但仍然会自动调用 Circle 类对应的无参构造函数，将基类成员半径 r 初始化为 0。然后构造函数的函数体将新增成员边框宽度 w 的初始值也设为 0（或任意其他值）。

■ BorderCircle　obj2( obj );

该定义语句用已经存在的对象 obj 来初始化新建对象 obj2。根据实参—形参匹配原则，执行该语句将自动调用拷贝构造函数(3)来初始化 obj2。初始化列表将形参 rObj 所引用的实参 obj 接力传递给 Circle 类对应的拷贝构造函数，用对象 obj 中的基类成员 r 来初始化新建对象 obj2 中的 r（半径）。然后构造函数的函数体将对象 obj 中的新增成员 w 赋值给新建对象 obj2 中的 w（边框宽度）。初始化后，新建对象 obj2 中的半径 r 和边框宽度 w 分别为 5 和 2，与原有对象 obj 完全一样。

#### 2. 派生类的析构函数

当对象生存期结束时，计算机销毁对象，释放其内存空间，这个过程就是对象的析构。销毁对象时计算机会自动调用其所属类的析构函数。类的析构函数只有一个，其功能是清

理内存，例如释放构造对象时所申请的额外内存。

派生类对象中数据成员的析构顺序是：先执行派生类析构函数的函数体，清理新增成员；再自动调用基类析构函数，清理基类成员。简单地说，派生类对象的析构顺序与构造顺序相反，即先析构新增成员，再析构基类成员。

### 3. 组合派生类的构造与析构

数据成员中包含对象成员的类称为组合类。如果派生类的新增成员中也包含对象成员，则该派生类就是**组合派生类**。组合派生类中的成员可分为三种：一是从基类继承来的成员（基类成员），二是新增的对象成员，三是新增的非对象成员。

为初始化对象，组合派生类的构造函数需依次初始化基类成员、新增对象成员和新增非对象成员。其中，初始化基类成员和新增对象成员需通过初始化列表，初始化新增的非对象成员则是在函数体中直接赋值。例 8-7 给出一个组合派生类的 C++示意代码。

**例 8-7　一个组合派生类的 C++示意代码**

**类 A1**

```
class   A1
{
public:
      int   a1;
      A1( int x = 0 )       // 构造函数
      {   a1 = x;   }
};
```

**类 A2**

```
class   A2
{
public:
      int   a2;
      A2( int x = 0 )       // 构造函数
      {   a2 = x;   }
};
```

**类 B1**

```
class   B1
{
public:
      int   b1;
      B1( int x = 0 )       // 构造函数
      {   b1 = x;   }
};
```

**类 B2**

```
class   B2
{
public:
      int   b2;
      B2( int x = 0 )       // 构造函数
      {   b2 = x;   }
};
```

**组合派生类 C**

```
class   C : public A1, public A2        // 继承基类 A1、A2
{
public:
      B1   bObj1;       // 类 B1 的对象成员 bObj1
      B2   bObj2;       // 类 B2 的对象成员 bObj2
      int   c;          // 非对象成员 c
      // 组合派生类的构造函数：初始化基类成员、新增对象成员、新增非对象成员
      C( int p1=0, int p2=0, int p3=0, int p4=0, int p5=0 ) : A1(p1), A2(p2), bObj1(p3), bObj2(p4)
      {   c = p5;   }
};
```

请注意初始化列表中的一个语法细节：初始化基类成员使用的是类名，而初始化新增对象成员使用的则是对象名。上述组合派生类 C 总共包含 5 个数据成员，分别是从基类继承的 a1 和 a2、新增对象成员的下级成员 bObj1.b1 和 bObj2.b2，以及新增的非对象成员 c，因此使用派生类 C 定义对象时需给出 5 个初始值，例如：

```
C   cObj( 2, 4, 6, 8, 10 );
```

该语句定义一个 C 类对象 cObj，其中的 5 个数据成员分别被初始化为 2、4、6、8、10。

执行组合派生类对象的定义语句，计算机将自动调用构造函数进行初始化。派生类对象中各成员的初始化顺序依次是：先调用基类构造函数初始化基类成员，再调用对象成员所属类的构造函数初始化新增对象成员，最后执行派生类构造函数的函数体初始化新增非对象成员。派生类对象中各成员的析构顺序与其构造顺序相反，即先析构新增的非对象成员，再析构新增对象成员，最后才析构基类成员。

## 8.3.5 继承与组合的应用

### 1. 重用类代码

重用已有的类代码可以减少重复开发，同时也能提高软件质量。面向对象程序设计有三种重用类代码的形式。

（1）**定义对象**。重用一个已有的类，可以用它来定义对象，然后访问对象的公有成员，以实现特定的程序功能。访问对象的数据成员，就是重用了类中的数据代码；调用对象的函数成员，就是重用了类中的算法代码。一个类可以定义多个对象，例如定义一个对象数组。

（2）**定义组合类**。重用一个已有的类，可以用它来定义组合类。可以使用同一个类定义出多种不同的组合类，即多种组合。所定义出的组合类可以继续用来定义更大的组合类，即多级组合。类的组合是一种“**自底向上**”的过程。

（3）**定义派生类**。重用一个已有的类，可以用它来定义派生类。可以继承同一个基类定义出多种不同的派生类，即多种派生。所定义出的派生类可以继续作为基类来定义更下级的派生类，即多级派生。类的继承与派生是一种“**自顶向下**”的过程。

图 8-11 给出了这三种重用类代码形式的示意图。

### 2. 继承与组合的区别

假设需要编写一个处理圆柱体的 C++程序，用于计算圆柱体的底面积、表面积和体积。程序员借助例 8-1 中的圆形类 Circle 来定义一个圆柱体类 Cylinder，可以采用组合或继承两种不同的方法。例 8-8 给出了分别使用组合和继承方法所编写出的计算圆柱体底面积、表面积和体积的 C++示例程序。

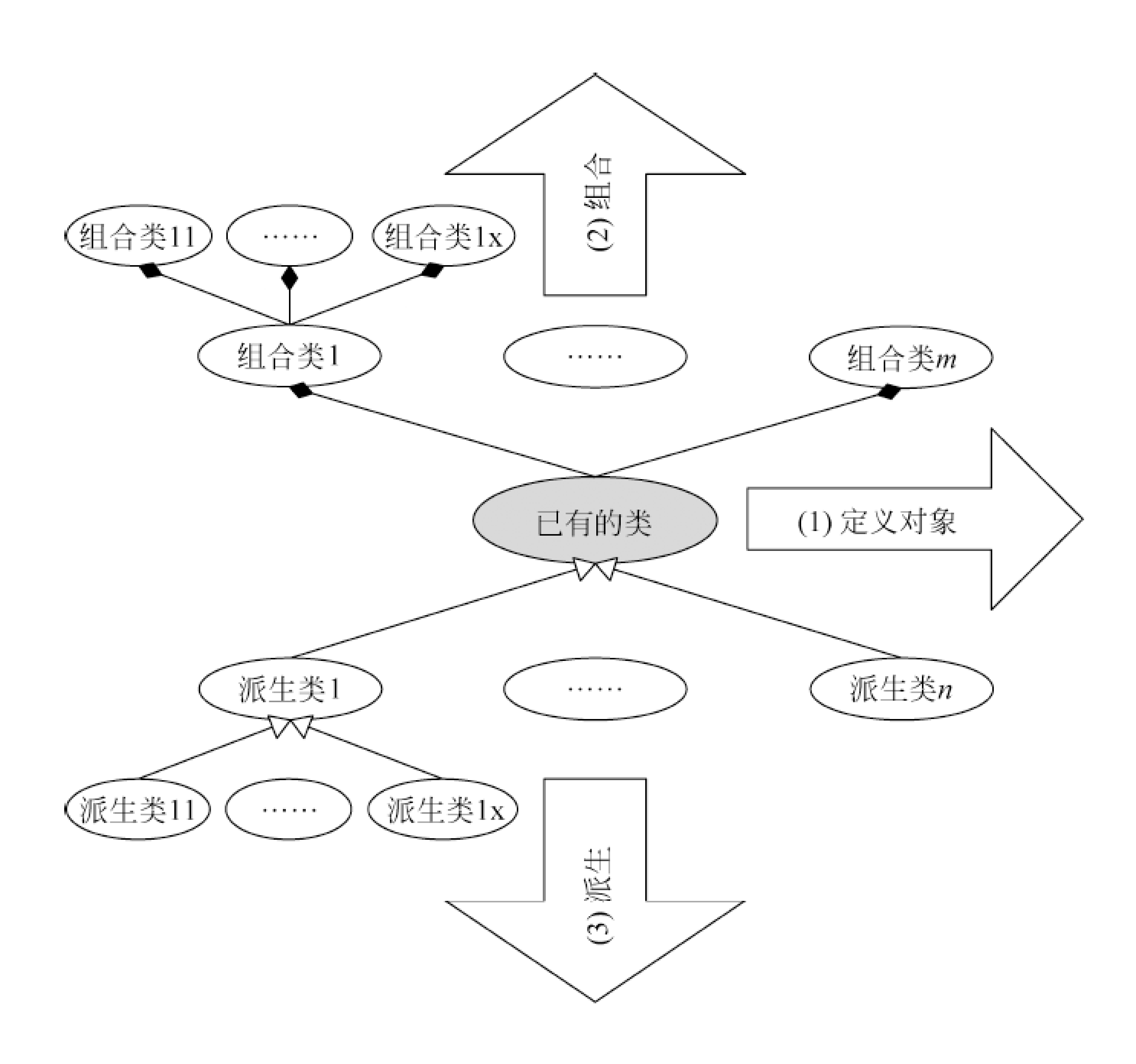



图 8-11　重用类代码的三种形式

**例 8-8　处理圆柱体的 C++示例程序**

**通过组合方法定义圆柱体类 Cylinder**

```
#include <iostream>
using namespace std;
#include "Circle.h"       // 声明类 Circle

class Cylinder            // 定义组合类 Cylinder
{
public:
    Circle  bottom;       // 声明底面 bottom
    double h;             // 声明圆柱体宽度 h
    void Input( )         // 输入半径和高度
    {
        bottom.Input( );  // 输入底面半径
        cin >> h;         // 输入高度
    }
    double Surface( )     // 求表面积
    {
        return  2*bottom.CArea( )
                + bottom.CLen( ) *h;
    }
    double Volumn( )      // 求体积
    {  return  bottom.CArea( ) *h;  }
};
```

**通过继承方法定义圆柱体类 Cylinder**

```
#include <iostream>
using namespace std;
#include "Circle.h"            // 声明类 Circle

// 改用继承的方法定义派生类 Cylinder
class Cylinder : public Circle
{
public:
    double h;                  // 声明圆柱体宽度 h
    void Input( )              // 输入半径和高度
    {
        Circle :: Input( );    // 输入底面半径
        cin >> h;              // 输入高度
    }
    double Surface( )          // 求表面积
    {
        return  2*CArea( )
                + CLen( ) *h;
    }
    double Volumn( )           // 求体积
    {  return  CArea( ) *h;  }
};
```

| | |
|---|---|
| int main( ) | int main( ) |
| { | { |
| Cylinder obj; // 定义组合类对象 obj | Cylinder obj; // 定义派生类对象 obj |
| obj.Input( ); // 输入半径和高度 | obj.Input( ); // 输入半径和高度 |
| cout << obj.bottom.CArea( ) << " "; | cout << obj. CArea( ) << " "; |
| cout << obj.Surface( ) << " "; | cout << obj.Surface( ) << " "; |
| cout << obj.Volumn( ) << endl; | cout << obj.Volumn( ) << endl; |
| return 0; | return 0; |
| } | } |

例 8-8 分别使用组合和继承方法来定义圆柱体类 Cylinder，进而编写出两个不同的程序。这两个程序的语法细节有较大不同，但功能完全相同，都是先输入底面半径和高度，然后计算并显示出圆柱体的底面积、表面积和体积。读者可对照这两个程序的代码，仔细体会组合和继承相关的语法细节。

既然组合和继承都可以实现程序功能，程序员到底该如何选择呢？读者仔细体会例 8-8 中的两个程序，会发现使用组合方法所编写的处理圆柱体程序更容易理解。是的，在客观世界中圆柱体就是由圆形底面和高度组合而成的，使用组合方法所定义出的圆柱体类 Cylinder 更加符合人们的认知。

使用一个已有的类来定义新类，假设已有的类代表客观事物 A，新类代表客观事物 B。如果 B 包含有（has）A，则 B 可认为是由 A 组合而成的，应当选择组合方法。如果 B 是（is）一种 A，则 B 可认为是 A 的一个特例，此时应当选择继承方法。

例如，圆柱体包含有圆形，使用圆形类 Circle 定义圆柱体类 Cylinder 时应当选择组合方法；而带边框的圆形是一种圆形，使用圆形类 Circle 定义带边框的圆形类 BorderCircle 时应当选择继承方法。

### 3. 凝练类代码

使用面向对象程序设计方法编写计算机程序，其设计过程是一个“自底向上，逐步抽象”的过程。程序员首先从实际问题中提取出一个个具体的客观对象，将具有共性的对象划分成类，然后用 C++语言描述该类的数据模型（即定义类），再按照类在内存中创建对应的内存对象（即定义对象）。程序通过访问内存对象中的公有成员来处理数据，最终实现特定的程序功能。

假设编写一个大学生信息管理系统的 C++程序，通过分析可抽象出本科生类 Undergraduate 和研究生类 Graduate 这两个类，其示意代码如例 8-9 所示。

**例 8-9　本科生类和研究生类的 C++示意代码**

| **Undergraduate**：本科生类 | **Graduate**：研究生类 |
|---|---|
| class Undergraduate // 声明部分 | class Graduate // 声明部分 |
| { | { |
| public: | public: |
| char Name[9], ID[11]; // 姓名、学号 | char Name[9], ID[11]; // 姓名、学号 |
| int Age; // 年龄 | int Age; // 年龄 |
| double Score; // 课堂成绩 | double Score1; // 课堂成绩 |
| double PracticeScore; // 毕业设计成绩 | double PaperScore; // 毕业论文成绩 |

```cpp
    void Input( );            // 输入学生信息
    void ShowInfo( );         // 显示学生信息
    double TotalScore( );     // 计算总成绩
};
```

```cpp
    int Thesis;               // 发表论文数量

    void Input( );            // 输入学生信息
    void ShowInfo( );         // 显示学生信息
    double TotalScore( );     // 计算总成绩
};
```

对比本科生类和研究生类，它们有很多共性的地方，例如姓名、学号、年龄、课堂成绩等都是一样的。这两个类也有一些区别，例如本科生有毕业设计成绩，而研究生则是毕业论文成绩以及所发表论文数量，由此会有不同的总成绩计算方法。

程序员可以在设计时将多个不同类中的共性部分进一步抽象出来形成基类，编码时再从基类进行派生，还原出各个不同的类。抽象出基类可以凝练类代码，有效减少程序中的重复代码。通过抽象基类可以改写例 8-9 中的类代码，改写后的示意代码如例 8-10 所示。

**例 8-10　通过抽象基类改写例 8-9 中类代码后的 C++示意代码**

**Student：** 抽象出的基类（学生类，类声明头文件 Student.h）

```cpp
class Student                 // 基类的声明部分
{
private:
    char Name[9], ID[11];     // 姓名、学号
    int Age;                  // 年龄
    double Score;             // 课堂成绩
public:
    void Input( );            // 输入学生基本信息
    void ShowInfo( );         // 显示学生基本信息
};
```

**Undergraduate：** 本科生派生类

```cpp
#include "Student.h"   // 声明基类 Student
class Undergraduate : public Student
{
public:
    double PracticeScore;   // 毕业设计成绩

    double TotalScore( );   // 计算总成绩
    void Input( );   // 输入：同名覆盖
    void ShowInfo( );   // 显示：同名覆盖
};
```

**Graduate：** 研究生派生类

```cpp
#include "Student.h"   // 声明基类 Student
class Graduate : public Student
{
public:
    double PaperScore;   // 毕业论文成绩
    int Thesis;   // 发表论文数量
    double TotalScore( );   // 计算总成绩
    void Input( );   // 输入：同名覆盖
    void ShowInfo( );   // 显示：同名覆盖
};
```

## 本节习题

1. 继承基类得到新的派生类，派生类中将不包括基类的哪种成员？（　　）
   A. 基类中的私有数据成员　　B. 基类中的保护数据成员
   C. 基类中的公有数据成员　　D. 基类中的构造函数和析构函数

2. 已定义类 A：

```
class A
{
   private:   int x;
   protected:   int y;
   public:   int z;
      void ShowA( )   {   cout << x << y << z << endl;   }
};
```

通过继承定义派生类 B：

```
class B : public A
{
   private:   int a;
   public:
      void ShowB( )   {   cout << x;   cout << y;   cout << z;   cout << a;   }
};
```

函数 ShowB( )中错误的语句是（　　）。

A. cout << x;　　B. cout << y;　　C. cout << z;　　D. cout << a;

3. 定义习题 2 的派生类 B 的对象 obj，则下列访问对象 obj 成员的语句中，正确的是（　　）。

A. obj.x = 5;　　B. obj.y = 5;　　C. obj.z = 5;　　D. obj.a = 5;

4. 如将习题 2 的派生类 B 的继承方式改为 private，则下列访问 B 类对象 obj 成员的语句中，正确的是（　　）。

A. obj.x = 5;　　B. obj.y = 5;

C. obj.z = 5;　　D. 基类继承的成员都不能访问

5. 如将习题 2 的派生类 B 的继承方式改为 protected，则下列访问 B 类对象 obj 成员的语句中，正确的是（　　）。

A. obj.x = 5;　　B. obj.y = 5;

C. obj.z = 5;　　D. 基类继承的成员都不能访问

6. 通过派生类对象 obj 访问其从基类继承的成员 m，则 m 必须是（　　）。

A. 公有继承下的公有成员　　B. 公有继承下的保护成员

C. 公有继承下的私有成员　　D. 私有继承下的公有成员

7. 已定义基类 A 和派生类 B：

```
class A
{
   private:   int x;
   protected:   int y;
   public:   int z;
      void ShowA( )   {   cout << x << y << z << endl;   }
};
class B : protected A   //  保护继承
{
   private :   int a;
```

```
    protected:    int b;
    public:    int c;
        void ShowB( )    {    ShowA( );    cout << a << b << c << endl;    }
};
```

再定义 B 的派生类 C：

```
class C : public B
{
    public:    int m;
        void fun( )    {    x = 5;    y = 5;    z = 5;    ShowA( );    }    // 访问基类 A 的成员
};
```

函数 fun( )中错误的语句是（　　）。

A. x = 5;　　B. y = 5;　　C. z = 5;　　D. ShowA( );

8. 已定义类 A：

```
class A
{
    ptivate:    int x;
    protected:    int y;
    public:    int z;
        A(int p1, int p2, int p3)    {    x = p1;    y = p2;    z = p3;    }    // 构造函数
};
```

再定义派生类 B：

```
class B : public A
{
    private:    int a;
    public:
        // 定义派生类 B 的构造函数
};
```

则下列派生类 B 的构造函数定义中，正确的是（　　）。

A. B(int p1, int p2, int p3, int p4)　{　x = p1;　y = p2;　z = p3;　a = p4;　}

B. B(int p1, int p2, int p3, int p4)　{　A(p1, p2, p3);　a = p4;　}

C. B(int p1, int p2, int p3, int p4)　{　a = p4;　}

D. B(int p1, int p2, int p3, int p4) : A(p1, p2, p3)　{　a = p4;　}

## 8.4　多态性

多态性（polymorphism）也是一个生物学概念，是指生物会在不同层面上体现出形态的多样性。其中，遗传多态性又特指同一种群内不同子群体之间所发生的多态现象。

程序设计中，多态性在字面上可理解为是一种程序代码的多义性。面向对象程序设计之所以引入多态的思想，其目的仍然是为进一步提高程序代码的可重用性，进而提高软件

开发效率。

## 8.4.1 面向对象程序中的多态

C++语言中有一些多义词，例如表示静态的关键字 static，将 static 应用在变量、函数或类成员等不同场合，它所表达的含义是不一样的。C++源程序是程序员编写的下达给计算机执行的指令序列。在源程序中看起来相同的关键字、运算符或标识符，但在编译或执行时会根据上下文被作出不同的语法解释。

源程序中相同的程序元素可能会具有不同的语法解释，C++语言称这些程序元素具有**多态性**。C++语言有多种不同的多态形式，常见的有**关键字多态**、**重载函数多态**、**运算符多态**、**对象多态**和**参数多态**等。

对于源程序中具有多态性的程序元素，什么时候对它们作出最终明确的语法解释呢？任何下达给计算机的指令都必须在具有明确的语法解释后才能被计算机执行，否则不能执行。对具有多态性的程序元素作出最终明确的语法解释，这称为**多态的实现**。实现多态有两个时间点，分别是在程序编译的时候，或是在程序执行的时候。不同多态形式具有不同的实现时间点，编译时实现的多态称为**编译多态**，执行时实现的多态称为**执行多态**。

### 1. 关键字多态

C++语言中的某些关键字是多义词，具有多态性，例如 static、const、void 以及 public/private/protected 等。关键字多态是由编译器在编译源程序时根据上下文进行语法解释的，是一种编译多态。

### 2. 重载函数多态

如果两个函数的形参个数不同，或数据类型不同，那么这两个函数就可以重名，它们被称为重载函数。编译时，由编译器根据调用语句中实参的个数和类型自动调用形参最匹配的那个重载函数。相同的重载函数名，调用时可能会调用不同的函数，这就是重载函数多态。重载函数多态是由编译器在编译源程序时实现的，也是一种编译多态。

所谓实现重载函数多态，就是在编译时将调用语句中的函数名转换成对应重载函数的内存存储地址。将源程序中的函数名转换成某个具体的函数存储地址，这种函数名到存储地址的转换被称为是对函数的**绑定**（binding）。

本章下面的内容主要是讲解运算符多态和对象多态。参数多态留待 10.1.2 节再做讲解。

## 8.4.2 运算符的多态与重载

C++语言中的运算符具有多态性。例如在 C++源程序中，整数加法和实数（浮点数）加法使用的是同一个加法运算符“+”。

```
2 + 3          // 整数加法
2.5 + 3.5      // 浮点数加法
```

通常所说的运算器是指定点运算器，只能进行整数运算。而浮点数运算则是通过浮点运算器（或称为协处理器）完成的。相同的运算符，计算机会根据数据类型来选择执行不同的运算，这就是运算符的多态性。运算符多态是由编译器在编译时进行语法解释的，是一种编译多态。

C++语言预定义了四十多个运算符，但主要是对基本数据类型的数据进行运算。对于像“类”这样的自定义数据类型，又该如何进行运算呢？请看下面一个复数类 Complex 的示意代码：

```
class Complex
{
private:
    double   real, image;            // 实数的实部和虚部
public:
    Complex(double x=0, double y=0)  {   real = x;   image = y;  }          // 构造函数
    Complex(Complex &c)   {   real = c.real;   image = c.image;  }          // 拷贝构造函数
    void Show( )   {   cout << real << "+" << image << "i" << endl;   }      // 显示复数
} ;
```

使用复数类定义三个复数对象 c1、c2、c3：

```
Complex   c1(1, 3), c2(2, 4), c3;
```

可以对这些复数对象进行算术运算或关系运算吗？答案是肯定的，但需要程序员为复数类型重新定义运算符的运算规则，因为它是程序员自己定义的数据类型。重新定义 C++语言已有运算符的运算规则，使同一运算符作用于不同类型数据时执行不同的运算，这就是**运算符重载**。正是因为 C++语言支持运算符多态，程序员才能重载运算符，实现**类运算**。

重载运算符就是以函数的形式来重新定义运算符的运算规则。程序员定义运算符函数的一般语法形式为：

```
函数类型 operator 运算符( 形式参数 )
{     函数体     }
```

为类重载运算符，可以将运算符函数定义成类的函数成员，也可以定义成类外的一个友元函数。这两种方法所实现的功能相同，但定义时在形参和函数体实现部分会有一些差别。另外，针对不同的运算符，其运算符函数的具体实现方法也有所不同，例如单目/双目、前置/后置等。本节将以复数类 Complex 为例，给出几种具有代表性的运算符函数实现方法。

### 1. 双目运算符“+”

例 8-11 给出两种不同的为复数类重载双目运算符“+”的 C++示意代码，其中一种将运算符函数定义成复数类的函数成员，另一种定义成复数类的友元函数。

**例 8-11　为复数类重载双目运算符“+”的 C++示意代码**

**代码 1：**定义成复数类的**函数成员**

```
class Complex
{
private:
    double   real, image;
public:
    Complex(double x=0, double y=0)
    {   real = x;    image = y;   }
    Complex(Complex &c)
    {   real = c.real;    image = c.image;   }
    void Show( )
    {
      cout<<real<<"+"<<image<<"i"<<endl;
    }
    Complex   operator +( Complex c )
    {
        Complex result ;
        result.real = real + c.real ;
        result.image = image + c.image ;
        return result ;
    }
} ;
```

**代码 2：**定义成复数类的**友元函数**

```
class Complex
{
private:
    double   real, image;
public:
    Complex(double x=0, double y=0)
    {   real = x;    image = y;   }
    Complex(Complex &c)
    {   real = c.real;    image = c.image;   }
    void Show( )
    {
      cout<<real<<"+"<<image<<"i"<<endl;
    }
friend Complex operator +(Complex c1, Complex c2) ;
} ;

Complex   operator +( Complex c1, Complex c2 )
{
    Complex result ;
    result.real = c1.real + c2.real ;
    result.image = c1.image + c2.image ;
    return result ;
}
```

例 8-11 演示了两种不同的运算符函数定义形式，请注意它们在形参和函数体实现部分的细小差别。无论使用哪种形式，都是为复数类定义了复数的加法运算规则：实部与实部相加，虚部与虚部相加。可以看出，如果不将运算符函数定义成类中的函数成员，那它就是类外的普通函数。为了让类外的运算符函数能访问类中的非公有成员，就必须将它定义成类的友元函数。

在为复数类重载加法运算符之后，就可以对复数对象进行加法运算了。例如，

```
Complex   c1(1, 3), c2(2, 4), c3;
c3 = c1 + c2;                  // 将对象 c1+c2 的和赋值给 c3
c3.Show( );                    // 显示复数 c3，显示结果：3+7i
```

计算机执行“c1+c2”的加法运算相当于执行一次函数调用，其调用形式如下：

（1）若运算符“+”被重载为复数类的**函数成员**，则调用形式为“**c1.+( c2 )**”。其中 c1 是对象名，“.”是成员运算符，“+”是函数成员名，c2 是实参。

（2）若运算符“+”被重载为复数类的**友元函数**，则调用形式为“**+( c1, c2 )**”。其中“+”是友元函数名，c1 和 c2 是实参。

将运算符函数的形参改为引用形式可以提高函数执行效率。例如将例 8-11 代码 1 中“+”

运算符函数成员的函数头改写成如下的形式：

```
Complex    operator +( Complex &c )                    // 函数成员：将形参 c 改为引用
```

还可以进一步引入 const 数据保护机制，将形参 c 改为常引用，因为加法运算不会修改参与运算操作数的值。例如，

```
Complex    operator +( const Complex &c )              // 函数成员：将形参 c 改为常引用
```

### 2. 单目运算符“++”

例 8-12 为复数类重载了自增运算符“++”，其中定义两个函数成员来分别实现“++”的前置和后置，这两个函数成员在形参和函数体实现部分存在细小的差别。因为这两个函数的名字都是“++”。为了能够重载，C++语言规定：前置单目运算符重载为函数成员时没有形参，而后置单目运算符重载时需要有一个 int 型形参。这个 int 型形参没有参数名，在函数体中并不使用，其目的纯粹是为了使这两个重名函数具有不同的形参，这样才能够重载。

**例 8-12　为复数类重载单目运算符“++”的 C++示意代码**

```
class Complex
{
private:
    double    real, image;
public:
    Complex(double x=0, double y=0)
    {    real = x;    image = y;    }
    Complex(Complex &c)
    {    real = c.real;    image = c.image;    }
    void Show( )
    {    cout<<real<<"+"<<image<<"i"<<endl;    }
    Complex &    operator ++( )            // 实现前置“++”运算符的函数成员
    {
        real ++;       image++;            // 运算规则：实部与虚部各加 1
        return    *this ;   // 返回前置“++”表达式的结果：当前对象加 1 之后的引用
    }
    Complex    operator ++( int )          // 实现后置“++”运算符的函数成员
    {
        Complex temp( *this );
        real ++;       image++;            // 运算规则：实部与虚部各加 1
        return    temp ;   // 返回后置“++”表达式的结果：加 1 之前的对象 temp
    }
};
```

在为复数类重载自增运算符“++”之后，就可以对复数对象进行自增运算了。

（1）前置自增运算，例如：

```
Complex    c1(1, 3), c2;
```

```
c2 = ++c1;                  // 前置自增运算
c1.Show( );                 // 显示复数 c1，实部与虚部各加 1。显示结果：2+4i
c2.Show( );                 // 显示复数 c2，即 c1 自增之后的结果。显示结果：2+4i
```

（2）后置自增运算，例如：

```
Complex   c1(1, 3), c2;
c2 = c1++;                  // 后置自增运算
c1.Show( );                 // 显示复数 c1，实部与虚部各加 1。显示结果：2+4i
c2.Show( );                 // 显示复数 c2，即 c1 自增之前的结果。显示结果：1+3i
```

### 3. 关系运算符“==”

如果将关系运算符“==”重载为复数类的函数成员，其示意代码如下：

```
bool   Complex :: operator ==( Complex c )           // 关系运算：检查两个复数是否相等
{
      return ( real == c.real && image == c.image ) ;  // 运算规则：实部与虚部是否都相等
}
```

需要注意的是，关系运算符函数的返回结果为 bool 型。在为复数类重载了关系运算符“==”之后，就可以对复数对象进行如下的关系运算：

```
Complex   c1(1, 3), c2(2, 4);
if (c1 == c2)                          // 比较 c1、c2 是否相等，比较结果为 false
      // 代码省略
```

### 4. 赋值运算符“=”

对象可以用赋值运算符“=”进行赋值。如果将赋值运算符重载为复数类的函数成员，其示意代码如下：

```
Complex &   Complex :: operator =( Complex c )
{
     real = c.real ;    image = c.image ;         // 一一对应地复制数据成员
     return    *this ;                            // 返回赋值后当前对象的引用
}
```

在为复数类重载了赋值运算符“=”之后，就可以对复数对象进行如下的赋值运算：

```
Complex   c1(1, 3), c2;
c1.Show( );                            // 显示复数 c1，显示结果：1+3i
c2 = c1;                               // 将对象 c1 赋值给 c2，即一一对应地拷贝数据成员
c2.Show( );                            // 显示复数 c2，结果与 c1 相同：1+3i
```

为了方便程序员，C++编译器通常会自动为类重载赋值运算符“=”。程序员也可以自己定义赋值运算符函数，其功能很像拷贝构造函数，就是把“=”右边表达式结果（应为一个对象）的数据成员一一对应地拷贝给左边被赋值对象的数据成员。

如果某个类在构造函数中动态分配了的内存，那么就需要为该类编写析构函数来释放

这些内存。此外，拷贝构造函数和重载赋值运算符“=”的函数也都需要程序员自己来编写。其目的是进行深拷贝，为新建对象或被赋值对象动态再分配同样多的内存。

### 5. 右移运算符“>>”和左移运算符“<<”

可以为类重载右移运算符“>>”和左移运算符“<<”，其目的是为了直接使用 cin 和 cout 指令来输入或输出对象。重载右移运算符“>>”和左移运算符“<<”时，只能将它们重载为类外的友元函数。例如，为复数类重载右移运算符“>>”和左移运算符“<<”，其示意代码如下：

```
class Complex
{
private:
    double    real, image;
public:
    Complex(double x=0, double y=0)
    {    real = x;    image = y;    }
    friend istream & operator >>( istream &is, Complex &c);            // 声明为友元函数
    friend ostream & operator <<( ostream &os, const Complex &c );
};
istream & operator >>( istream &is, Complex &c)              // 重载右移运算符“>>”
{
    is >> c.real >> c.image;                                  // 通过键盘输入复数的实部和虚部
    return is;
}
ostream & operator <<( ostream &os, const Complex &c )       // 重载左移运算符“<<”
{
    os << c.real << "+" << c.image << "i";                    // 在显示器上显示复数
    return os;
}
```

注：上述代码中的 istream 和 ostream 将在后面的第 9.2 节中再做介绍。

在为复数类重载了右移运算符“>>”和左移运算符“<<”之后，就可以直接使用 cin 和 cout 指令来输入输出复数对象。例如：

```
Complex    cObj;
cin >> cObj;                          // 假设通过键盘输入复数的实部和虚部：3 5<回车>
cout << cObj << endl;                 // 将在显示器上显示：3+5i
```

运算符重载的语法细则：

（1）除了下面的 5 个运算符，C++语言中的其他运算符都可以重载。这 5 个不能重载的运算符是：条件运算符“?:”、sizeof 运算符、成员运算符“.”、指针运算符“*”和作用域运算符“::”。

（2）重载后，运算符的优先级和结合性不会改变。

（3）重载后，运算符的操作数个数不能改变，同时至少要有一个操作数是类类型。
（4）重载后，运算符的含义应与原运算符相似，否则会给使用者造成困惑。

### 本节习题

1. 下列哪种 C++语法形式不属于多态？（ ）
   A. 大写和小写字母 B. 重载函数 C. 重载运算符 D. 对象多态
2. 下列关于多态性的描述中，错误的是（ ）。
   A. 源程序中相同的程序元素可能会有不同的语法解释，这就是程序元素的多态性
   B. 对具有多态性的程序元素作出最终明确的语法解释，这称为多态的实现
   C. 所有形式的多态都是在编译时实现的
   D. 引入多态思想的目的是为进一步提高程序代码的可重用性
3. 下列关于运算符重载的描述中，错误的是（ ）。
   A. 重载运算符就是重新定义 C++语言中已有运算符的运算规则
   B. 运算符函数可定义成类的函数成员
   C. 运算符函数可定义成类外的友元函数
   D. 定义运算符函数必须使用关键字 friend
4. 为类 ABC 定义重载运算符“+”，下列哪种定义形式是正确的？（ ）
   A. 定义为类 ABC 的函数成员：void operator +( ) { … }
   B. 定义为类 ABC 的函数成员：ABC operator +( ABC obj ) { … }
   C. 定义为类 ABC 的函数成员：ABC operator +( ABC obj1, ABC obj2 ) { … }
   D. 定义为类 ABC 的友元函数：ABC operator +( ABC obj ) { … }

# 8.5 对象的替换与多态

除构造函数、析构函数之外，派生类将继承基类中的所有数据成员和函数成员。派生类和基类之间存在这样一种特殊的关系：派生类是一种基类，具有基类的所有功能。面向对象程序设计利用派生类和基类之间的这种特殊关系，常常将派生类对象当作基类对象来使用，或者用基类来代表派生类，其目的是为了提高程序中算法代码的**可重用性**。

## 8.5.1 算法代码的可重用性

### 1. 什么是算法代码的可重用性

程序中最常见的算法代码形式是函数。假设已定义一个处理 int 型数据的函数 fun，让我们来分析一下该函数代码的可重用性。

```
void fun( int x )  {  cout << x*x;  }      // 计算并显示 x 的平方
```

调用函数 fun 可以处理 int 型数据，例如计算并显示 5 的平方。

```
fun( 5 );                    // 正确：调用函数时，实参的数据类型与形参一致
```

但是不能调用函数 fun 来处理 double 型数据，例如求实数 5.8 的平方。

```
fun( 5.8 );                  // 错误：5.8 是 double 类型，与函数 fun 中形参的类型不一致
```

**结论 1**：C++语言对数据类型一致性的要求比较严格，属于强类型检查的计算机语言。C 语言和 Java 语言也都属于强类型检查的语言。

因为数据类型不一致，不能重用函数 fun 的代码来处理 double 型数据。如需处理，程序员必须再编写一个处理 double 型数据的函数 fun，尽管函数中求平方的算法代码完全一样。例如：

```
void fun( double x )  {  cout << x*x;  }  // 处理 double 型数据的重载函数 fun
```

再进一步，如果已经定义了一个类 A 和一个处理 A 类对象的函数 aFun，我们来分析一下处理对象数据（即类类型数据）函数代码的可重用性问题。

```
class A  {  …  };                  // 定义一个类 A
void aFun( A x )  {  …  }          // 处理 A 类对象的函数 aFun，不必关注具体的算法
```

可以调用函数 aFun 来处理 A 类对象，例如：

```
A   aObj;                          // 定义一个 A 类对象 aObj
aFun( aObj );                      // 正确：调用函数时，实参的数据类型与形参一致
```

假设还有一个类 B：

```
class B  {  …  };                  // 定义一个类 B
```

能否用函数 aFun 来处理 B 类的对象呢？例如：

```
B   bObj;                          // 定义一个 B 类对象 bObj
aFun( bObj );                      // 错误：bObj 的类型与函数 aFun 中形参的类型不一致
```

**结论 2**：不能调用处理 A 类对象的函数 aFun 来处理 B 类的对象数据。

在面向对象程序设计中，重用处理基类对象的算法代码来处理派生类对象，这是非常普遍的需求。如果派生类对象能够与基类对象共用算法代码，它将极大地提高程序代码的重用性。为此，面向对象程序设计方法利用派生类和基类之间存在的特殊关系，提出了对象的替换与多态。例如，如果类 B 公有继承类 A：

```
class B : public A  {  …  };       // 类 B 公有继承类 A，是类 A 的派生类
```

则可以用处理 A 类对象的函数 aFun 来处理 B 类对象，即派生类对象与基类对象共用算法代码。

### 2. 直观认识对象的替换与多态

这里通过一个程序例子来直观认识一下什么是对象的替换和多态。

```
class A                                        // 定义基类 A
{
public:
    void fun1( )                               // 普通的函数成员 fun1
    {   cout << "A::fun1 called\n";   }        // 调用时将显示类 A 的 fun1 被调用
    virtual void fun2( )                       // 加关键字 virtual 表示 fun2 是一个虚函数成员
    {   cout << "Virtual A::fun2 called\n";   } // 调用时将显示类 A 的 fun2 被调用
};
class B : public A                             // 定义派生类 B，公有继承 A
{
public:
    void fun1( )                               // 又新增一个同名的普通函数成员 fun1
    {   cout << "B::fun1 called\n";   }        // 调用时将显示类 B 的 fun1 被调用
    virtual void fun2( )                       // 再新增一个同名的虚函数成员 fun2
    {   cout << "Virtual B::fun2 called\n";   } // 调用时将显示类 B 的 fun2 被调用
};
```

这样派生类 B 中将包含 4 个函数成员，分别是从类 A 中继承来的基类成员 A::fun1 和 A::fun2，以及新增的函数成员 B::fun1 和 B::fun2。其中的两个 fun1 是普通函数成员，另外两个 fun2 是虚函数成员。

下面我们进行一个对象替换与多态的程序实验。

**第 1 步：**先定义对象。

```
A   aObj;                  // 定义一个 A 类的对象 aObj
B   bObj;                  // 定义一个 B 类的对象 bObj
```

**第 2 步：**再定义两个基类引用 ra1 和 ra2，分别引用基类对象 aObj 和派生类对象 bObj。

```
A   &ra1 = aObj, &ra2 = bObj;      // ra1 引用基类对象 aObj，ra2 引用派生类对象 bObj
```

**第 3 步：**通过基类引用 ra1、ra2 访问基类对象 aObj 和派生类对象 bObj，分别调用它们的普通函数成员 fun1，观察调用结果。

```
ra1.fun1( );               // 将调用基类对象 aObj 中的函数成员 fun1，显示：A::fun1 called
ra2.fun1( );               // 将调用派生类对象 bObj 中的基类成员 A::fun1，显示：A::fun1 called
```

观察发现，通过基类引用来调用普通函数成员 fun1，基类对象 aObj 和派生类对象 bObj 都显示出了信息“**A**::fun1 called”，它们的调用结果一样。换句话说，接收相同的指令 fun1，派生类对象 bObj 所表现出的行为和基类对象 aObj 也一样，这时的 bObj 就像是一个基类对象。通过基类引用 ra2 来引用派生类对象 bObj，相当于是将派生类对象当作基类对象来使用，这被称作**对象替换**。C++语言中的对象替换需遵循**类型兼容语法规则**。

**第 4 步：**再通过基类引用 ra1、ra2 来访问基类对象 aObj 和派生类对象 bObj，分别调用它们的虚函数成员 fun2，观察调用结果。

```
ra1.fun2( );        // 将调用基类对象 aObj 中的虚函数成员 fun2，显示：Virtual A::fun2 called
ra2.fun2( );        // 将调用派生类对象 bObj 中的新增虚函数成员 fun2，显示：Virtual B::fun2 called
```

观察发现，通过基类引用来调用虚函数成员 fun2，基类对象 aObj 和派生类对象 bObj 会显示出不同的信息（体现在“A::fun2”和“B::fun2”上）。换句话说，接收相同的指令 fun2，基类对象 aObj 和派生类对象 bObj 表现出了不同的行为，呈现出多样化的形态，这就是对象的**多态性**。

## 8.5.2　钟表类及其处理算法

本节通过一个关于钟表的程序实例，重点解释为什么派生类对象需要与基类对象一起共用算法代码。

### 1. 钟表类 Clock 举例

一个钟表类 Clock 应包含 3 个数据成员，分别为时、分、秒（假定为北京时间）。为简单起见，可只定义 2 个函数成员，分别用于设置时间和显示时间。例 8-13 给出钟表类 Clock 的示意代码。

**例 8-13　钟表类 Clock 的 C++示意代码（类声明头文件名 Clock.h）**

```
class Clock                                  // 钟表类 Clock
{
private:
    int   hour, minute, second;              // 时、分、秒：北京时间
public:
    void Set( int h, int m, int s )          // 设置时间
    {   hour = h;   minute = m;   second = s;   }
    void Show( )                             // 显示时间
    {   cout << hour << ":" << minute << ":" << second;   }
} ;
```

可以定义 Clock 类的钟表对象 obj，调用其公有函数成员实现设置和显示时间的功能。例如：

```
Clock   obj;                  // 定义一个 Clock 类的钟表对象 obj
obj.Set( 8, 30, 0 );          // 将钟表对象 obj 的时间设为 8 点 30 分（北京时间）
obj.Show( );                  // 显示钟表对象 obj 的时间，显示结果：8:30:0
```

### 2. 处理钟表类对象的算法

在应用钟表类 Clock 的过程中可能会遇到各种各样的实际需求，下面给出两个例子。

（1）**GMT 时间**。钟表类 Clock 内部保存的是北京时间，实际应用中程序员可能只知道格林尼治时间（GMT，或称为世界时 UT）。可以定义如下的一个函数 SetGMT，将格林尼治时间转换成北京时间。

```
void SetGMT( Clock &rObj,   int hGMT, int mGMT, int sGMT )   // 设置钟表对象的时间
{
    int hBeijing = hGMT +8;                  // 北京时间要早 8 个小时，需将小时数加 8
```

```
        hBeijing = hBeijing % 24;              // 小时数应在 0~23 之间
        rObj.Set( hBeijing, mGMT, sGMT );      // 将北京时间设置给钟表对象
    }
```

函数 SetGMT 的第一个形参 rObj 是一个 Clock 类的引用，后面三个形参用于接收 GMT 时间。使用 SetGMT 函数，程序员就可以在设置钟表时给 GMT 时间，由 SetGMT 函数将 GMT 时间转换成北京时间。例如：

```
Clock    obj;                    // 定义一个 Clock 类的钟表对象 obj
SetGMT( obj, 8, 30, 0 );         // 将钟表对象 obj 的时间设为 8 点 30 分（GMT 时间）
// 8 点 30 分（GMT 时间）转换后的北京时间是 16 点 30 分，比 GMT 时间早 8 个小时
obj.Show( );                     // 显示钟表对象 obj 的时间（北京时间），显示结果：16:30:0
```

（2）**明确时间的时区**。在定义了 SetGMT 函数之后，程序中就出现了 GMT 时间和北京时间这样两种时间。为了明确表明钟表对象内部保存的是北京时间，程序员可以再定义一个新的显示时间函数 ShowBJT，在所显示的时间前面加上“BJT”这样一个前缀标记。

```
void ShowBJT( Clock &rObj )
{
    cout << "BJT-";              // 在时间前面加上“BJT”，明确表明是北京时间
    rObj.Show( );                // 再调用钟表对象的函数成员 Show 显示时间
}
```

程序员可以使用 ShowBJT 函数显示钟表对象的时间，所显示的时间前面都被加上了一个前缀“BJT”。例如：

```
Clock    obj;                    // 定义一个 Clock 类的钟表对象 obj
SetGMT( obj, 8, 30, 0 );         // 将钟表对象 obj 的时间设为 8 点 30 分（GMT 时间）
ShowBJT( obj );                  // 显示钟表对象 obj 的时间（北京时间），显示结果：BJT-16:30:0
```

在这个例子中，如果程序员不使用 ShowBJT 函数，而是直接调用钟表对象的函数成员 Show 来显示时间，那么所显示出的时间就没有前缀“BJT”。例如，

```
obj.Show( obj );                 // 显示钟表对象 obj 的时间（北京时间），显示结果：16:30:0
```

请读者记住，这里出现了两种代码，一是钟表类 Clock 的类代码，二是处理钟表类对象的函数代码 SetGMT 和 ShowBJT（属于算法代码）。C++语言的目标是这两种代码都要重用，这样效率才高。

### 3. 重用钟表类 Clock 的类代码

在应用钟表类 Clock 的过程中，程序员可以通过继承的方法定义各种各样的派生类，例如手表类 Watch、挂钟类 WallClock 等，还可以继续基于手表类 Watch 定义出潜水表类 DivingWatch。例 8-14 给出上述三个派生类的示意代码。

例 8-14　钟表类 Clock 的三个派生类示意代码

**手表类**：类头文件名 Watch.h

```
#include "Clock.h"   // 声明基类 Clock
class Watch : public Clock   // 公有继承
{
public:
    int band;   // 表带类型
    void Show( )   // 重写 Show：扩充功能
    {
        Clock :: Show( );
        cout << "（手表）";   // 增加类标签
    }
} ;
```

**挂钟类**：类头文件名 WallClock.h

```
#include "Clock.h"   // 声明基类 Clock
class WallClock : public Clock   // 公有继承
{
public:
    int size;   // 表盘尺寸
    void Show( )   // 重写 Show：扩充功能
    {
        Clock :: Show( );
        cout << "（挂钟）"; // 增加类标签
    }
} ;
```

**潜水表类**：类头文件名 DivingWatch.h

```
#include "Watch.h"   // 声明基类 Watch
class DivingWatch : public Watch // 公有继承
{
public:
    int depth;   // 最大下潜深度
    void Show( )   // 重写 Show：扩充功能
    {
        Clock :: Show( );
        cout << "（潜水表）"; // 增加类标签
    }
} ;
```

例 8-14 的程序说明如下：

- **手表类** Watch。手表是一种钟表，具有钟表的所有功能。手表在钟表的基础上增加表带类型（例如皮革或金属表带）。手表类公有继承钟表类，然后增加一个数据成员 band，再增加一个显示时间的函数成员 Show。新增的函数成员 Show 覆盖同名的基类成员，在所显示的时间后面添加一个类标签“手表”。
- **挂钟类** WallClock。挂钟也是一种钟表，具有钟表的所有功能。挂钟在钟表的基础上增加表盘尺寸的属性（例如 10 英寸、12 英寸或 14 英寸）。挂钟类公有继承钟表类，然后增加一个数据成员 size，另外也增加了一个显示时间的函数成员 Show，在所显示的时间后面添加一个类标签“挂钟”。
- **潜水表类** DivingWatch。潜水表在手表的基础上继续扩展，增加最大下潜深度的属性。潜水表类公有继承手表类，它是钟表类下的二级派生类。潜水表类在手表类的基础上再增加一个数据成员 depth，同时还增加一个显示时间的函数成员 Show，在所显示的时间后面添加一个类标签“潜水表”。

请读者关注这样一个**重要细节**：这三个派生类都继承了基类的 Set 函数成员，即原封不动地保留了基类的设置时间功能，但都新增了自己新的显示时间函数 Show，在时间的后面添加一个表明钟表类型的标签。这意味着派生类扩充了基类的显示时间功能，派生类对象将使用新的显示时间功能。例如：

```
Watch  wObj;              // 定义一个 Watch 类的手表对象 wObj
wObj.Set( 8, 30, 0 );     // 使用基类继承来的 Set 函数设置手表对象 wObj 的时间
wObj.Show( );             // 使用新的 Show 函数显示手表对象 wObj 的时间：8:30:0（手表）
DivingWatch  dObj;        // 定义一个 DivingWatch 类的潜水表对象 dObj
dObj.Set( 8, 30, 0 );     // 使用基类继承来的 Set 函数设置潜水表对象 dObj 的时间
dObj.Show( );             // 使用新的 Show 函数显示潜水表对象 dObj 的时间：8:30:0（潜水表）
```

类的继承和派生可以任意多级。用基类定义派生类，派生类可以继续作为基类去定义更下级的派生类，这就是多级派生。经过多级派生后，基类及其下面的各级派生类共同组成了一个具有继承关系和共同特性的类的家族，我们称之为**类族**。类族中的子类具有共同的祖先，都继承了基类中的成员。例如钟表类与手表类、挂钟类、潜水表类共同组成一个钟表的类族，它们都具有钟表的功能，其中钟表类 Clock 是大家的共同祖先。图 8-12 给出了钟表类族的继承关系图。

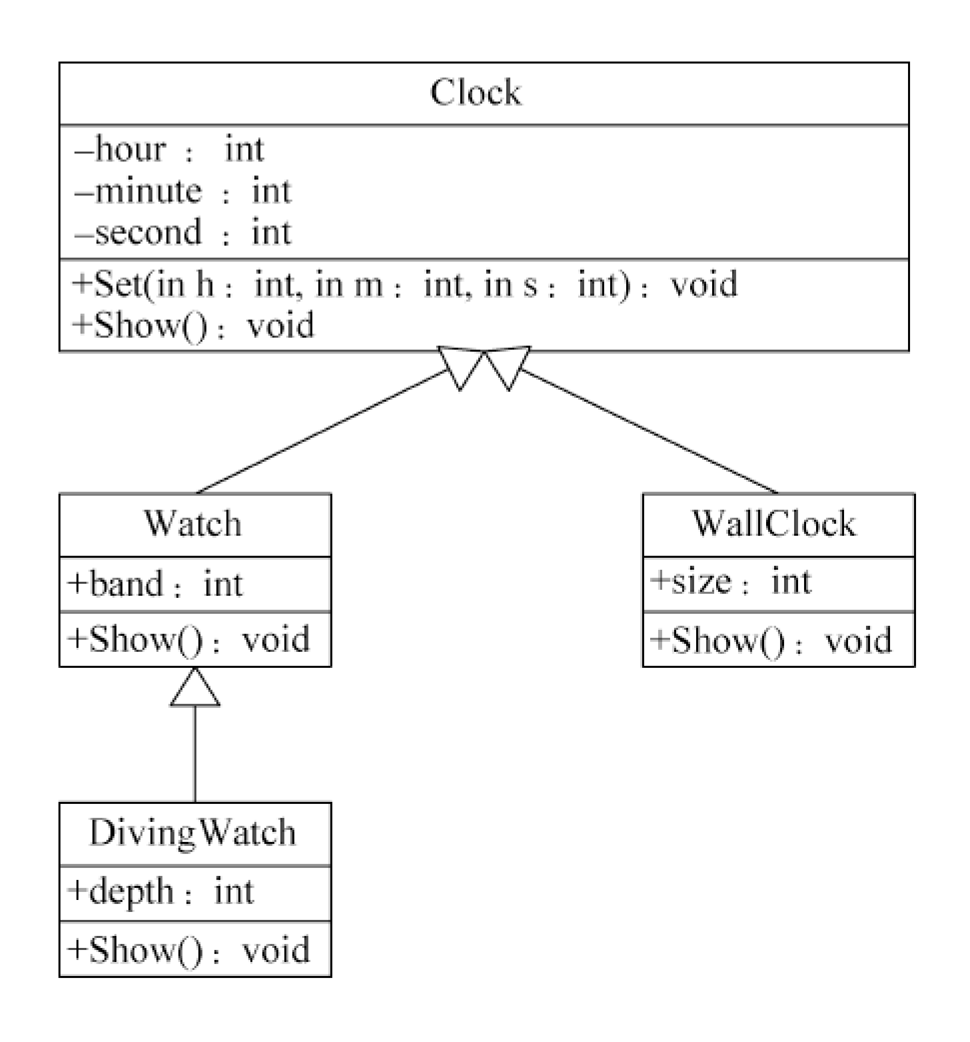



图 8-12 钟表类族的继承关系图

### 4. 重用处理 Clock 类对象的算法代码

在之前应用钟表类 Clock 的过程中，程序员还编写了两个处理钟表类对象的函数 SetGMT 和 ShowBJT。使用 SetGMT 函数，程序员可以在设置钟表时给出 GMT 时间，由 SetGMT 函数转换成北京时间。使用 ShowBJT 函数显示钟表对象的时间，所显示的时间前面都被加上了一个表明北京时间的前缀“BJT”。例如：

```
Clock  obj;                  // 定义一个 Clock 类的钟表对象 obj
SetGMT( obj, 8, 30, 0 );     // 将钟表对象 obj 的时间设为 8 点 30 分（GMT 时间）
ShowBJT( obj );              // 显示钟表对象 obj 的时间（北京时间），显示结果：BJT-16:30:0
```

函数 SetGMT 和 ShowBJT 是处理 Clock 类（基类）对象的算法代码。

在派生类重用了基类 Clock 的类代码之后，如果能让派生类对象继续重用处理基类对象的算法代码，这样就能更进一步地提高程序代码的重用性。为此，面向对象程序设计方

法利用派生类和基类之间存在的特殊关系，提出了**类型兼容语法规则**和**对象多态性**这两个重要概念。

### 8.5.3　类型兼容语法规则

本节以前面的 SetGMT 函数重用问题为例来具体讲解类型兼容语法规则，该函数的原型声明如下：

```
void SetGMT( Clock &rObj,   int hGMT, int mGMT, int sGMT );
```

函数 SetGMT 的功能是设置基类（即 Clock 类）对象的 GMT 时间。本节讨论如何重用这个函数来设置派生类（手表类 Watch、挂钟类 WallClock 和潜水表类 DivingWatch）对象的 GMT 时间。例如，使用 SetGMT 函数来设置 Watch 类对象的 GMT 时间。

```
Watch   wObj;                    // 定义一个派生类 Watch 的对象 wObj
SetGMT( wObj, 8, 30, 0 );        // 将派生类对象 wObj 的时间设为 8 点 30 分（GMT 时间）
```

**讨论点 1**：如果可以继续使用函数 SetGMT 给派生类对象设置 GMT 时间，那就意味着函数 SetGMT 的代码被重用了，这是程序员所期望的。

**讨论点 2**：函数 SetGMT 的形参 rObj 是一个基类 Clock 的引用，它应该只能引用基类 Clock 的对象。而这里调用 SetGMT 函数时所传递的实参 wObj 是一个派生类 Watch 的对象。对比形参和实参的类型，两者类型不一致。形实结合的结果是用一个基类引用 rObj 去引用一个派生类对象 wObj。原则上，这应当是不允许的。

#### 1. 类型兼容语法规则

为了让派生类对象可以与基类对象一起共用算法代码，C++语言专门制定了如下的**类型兼容语法规则**：

（1）派生类的对象可以赋值给基类对象。

（2）派生类的对象可以初始化基类引用，或者说基类引用可以引用派生类对象。

（3）派生类对象的地址可以赋值给基类的对象指针，或者说基类的对象指针可以指向派生类对象。

应用类型兼容语法规则有一个前提条件和一个使用限制。

**前提条件**：派生类必须公有继承基类。

**使用限制**：通过基类的对象、引用或对象指针访问派生类对象时，只能访问到基类成员。

类型兼容语法规则来源于 1987 年美国科学家 Liskov 所提出的 Liskov **替换准则**（Liskov Substitution Principle，LSP）。该准则的原文是“Inheritance should ensure that any property proved about supertype objects also holds for subtype objects”，即“继承必须确保基类对象所拥有的性质在派生类对象中依然成立”。在公有继承的情况下，派生类拥有基类的全部功能，派生类对象可以当作基类对象使用。

### 2. 语法演示

下面的程序代码完整演示了类型兼容语法规则，代码分别通过基类 Clock 的对象、引用或对象指针来访问派生类 Watch 的对象 wObj，并通过注释给出了相应的访问结果。读者比对这些结果就可以准确把握类型兼容语法规则的语法细节。

```
Watch   wObj;                // 定义一个手表类 Watch 的对象 wObj
wObj.Set( 8, 30, 0 );        // 将手表对象 wObj 的时间设为 8 点 30 分（北京时间）
wObj.Show( );                // 访问派生类对象 wObj 的成员，访问到的是新增成员 Show
                             // 新增成员 Show 所显示的时间含有类标签：8:30:0（手表）
wObj.band = 1;               // 设置手表对象 wObj 的表带类型，假设 1 表示皮革表带

// 演示 1：派生类的对象可以赋值给基类对象
Clock   obj = wObj;          // 将派生类对象 wObj 赋值给基类对象 obj
obj.Show( );                 // 访问赋值后基类对象 obj 的成员，访问到的是基类成员 Show
                             // 基类成员 Show 所显示的时间无类标签：8:30:0
cout << obj.band;            // 错误：赋值后基类对象中不包含派生类对象的新增成员 band

// 演示 2：通过基类引用访问派生类对象
Clock   &rObj = wObj;        // 定义基类引用 rObj，引用派生类对象 wObj
rObj.Show( );                // 通过基类引用 rObj 将调用基类成员 Show，时间无类标签：8:30:0
cout << rObj.band;           // 错误：通过基类引用访问派生类对象访问不到新增成员 band

// 演示 3：通过基类对象指针访问派生类对象
Clock   *pObj = &wObj;       // 定义基类指针对象 pObj，指向派生类对象 wObj
pObj->Show( );               // 通过基类对象指针 pObj 将调用基类成员 Show，时间无类标签：8:30:0
cout << pObj->band;          // 错误：通过基类对象指针访问派生类对象访问不到新增成员 band
```

注：上述代码对另外两个派生类（挂钟类 WallClock 和潜水表类 DivingWatch）同样适用。

### 3. 重用 SetGMT 函数

重新回顾一下 SetGMT 函数的定义：

```
void SetGMT( Clock &rObj,   int hGMT, int mGMT, int sGMT )
{
    int hBeijing = hGMT +8;                  // 北京时间要早 8 个小时，需将小时数加 8
    hBeijing = hBeijing % 24;                // 小时数应在 0~23 之间
    rObj.Set( hBeijing, mGMT, sGMT );        // 将北京时间设置给钟表对象
}
```

类型兼容语法规则允许通过基类引用来访问派生类对象，这样 SetGMT 函数就可以被用来设置派生类对象的 GMT 时间了，例如设置 Watch 类对象的时间。

```
Watch   wObj;                    // 定义一个派生类 Watch 的对象 wObj
SetGMT( wObj, 8, 30, 0 );        // 将派生类对象 wObj 的时间设为 8 点 30 分（GMT 时间）
```

在执行 SetGMT 函数调用语句时，形参基类引用 rObj 将引用实参派生类对象 wObj，然后在函数体中将 GMT 时间转换成北京时间，最后通过基类引用 rObj 调用基类成员 Set 来设置派生类对象 wObj 的时间。

GMT 时间 8 点 30 分对应的北京时间是 16 点 30 分。调用 Watch 类对象 wObj 的函数成员 Show 来显示时间，验证设置结果是否正确。

```
wObj.Show( );      // 显示 Watch 类对象 wObj 的时间，显示结果：16:30:0（手表）
```

显示结果正确。这表明函数 SetGMT 可以被重用，用于设置派生类对象的 GMT 时间。

总结一下，重用函数 SetGMT 需要满足两个前提条件，一是基类成员 Set 已经达到了派生类对象设置时间的功能要求，二是语法上可以通过基类引用去引用派生类对象。C++语言的类型兼容语法规则就是为满足上面的第二个条件而专门设计的。正是因为有了类型兼容语法规则，派生类对象才能和基类对象一起共用 SetGMT 函数这样的算法代码。

## 8.5.4　对象的多态性

本节以前面的 ShowBJT 函数重用问题为例来具体讲解对象的多态性。函数 ShowBJT 的功能是在显示钟表 Clock（基类）对象的时间前面加上一个前缀“BJT”，这样可以明确表明所显示的时间是北京时间。本节讨论如何重用这个函数来显示派生类（手表类 Watch、挂钟类 WallClock 和潜水表类 DivingWatch）对象的时间。例如，使用 ShowBJT 函数来显示 Watch 类对象的时间。

```
Watch   wObj;                    // 定义一个派生类 Watch 的对象 wObj
SetGMT( wObj, 8, 30, 0 );        // 将派生类对象 wObj 的时间设为 8 点 30 分（GMT 时间）
ShowBJT( wObj );                 // 显示 wObj 的时间，显示结果：BJT-16:30:0
```

结果表明，可以重用 ShowBJT 函数来显示派生类对象的时间。函数 ShowBJT 的形参 rObj 是一个基类 Clock 的引用，而这里调用函数所传递的实参 wObj 是一个派生类 Watch 的对象。形实结合的结果是基类引用 rObj 引用了一个派生类对象 wObj。之所以能够重用 ShowBJT 函数，其原理与上一小节的 SetGMT 函数一样，都是利用了 C++语言中的类型兼容语法规则。

### 1. 对象多态性问题的提出

再回顾派生类定义时的一个重要细节：钟表类 Clock 的派生类都新增了一个显示时间的函数成员 Show，在所显示的时间后面添加一个类标签。也就是说，派生类中有两个同名的函数成员 Show，一个是从基类 Clock 继承来的函数 Show，所显示的时间没有类标签；另一个是派生类新增的函数 Show，所显示的时间有类标签。派生类通过新增函数成员 Show 扩展了基类的显示时间功能。如果使用扩展后的显示时间功能，其显示结果与函数 ShowBJT 是不一样的，例如：

```
wObj.Show( );          // 调用 wObj 的新增函数成员 Show，显示结果：16:30:0（手表）
```

派生类 Watch 新增函数成员 Show 所显示的时间有一个后缀“（手表）”，它表示这个时间是一只手表的时间。而函数 ShowBJT 所显示的时间有一个表示北京时间的前缀“BJT”：**BJT**-16:30:0，但它没有显示出后缀“（手表）”。仔细分析下面的 ShowBJT 函数代码：

```
void ShowBJT( Clock &rObj )
```

```
{
    cout << "BJT-";          // 在时间前面加上“BJT”，明确表明是北京时间
    rObj.Show( );            // 再调用钟表对象的函数成员 Show 显示时间
}
```

对比发现，函数 ShowBJT 通过基类引用 rObj 来访问派生类对象 wObj 的函数成员 Show，只能访问到基类成员 Show（即类型兼容语法规则），所显示的时间没有类标签。而通过派生类对象 wObj 自己的对象名访问时，将访问到派生类的新增成员 Show（即同名覆盖），所显示的时间有类标签。

如果想同时保留前缀和后缀，显示出一个完整的时间，例如：**BJT**-16:30:0（**手表**），那该怎么办呢？在这种情况下程序员就需要使用对象多态技术，将函数成员 Show 定义成**虚函数**，然后在函数 ShowBJT 中继续通过基类引用 rObj 来访问派生类对象 wObj 的虚函数成员 Show。访问虚函数成员，C++语言能够根据实际所引用对象的类型，自动访问派生类中功能扩展以后的新增成员。

### 2. 类中函数成员的同名问题

对象多态性这个术语是从生物多态性借鉴而来的。例如，米老鼠、唐老鸭分别是老鼠类和鸭子类的对象。向米老鼠下达指令“Go”，米老鼠将迈开四条腿迅速移动。向唐老鸭下达指令“Go”，唐老鸭则迈开两条腿蹒跚而行。不同生物对象在执行相同指令“Go”的时候会表现出不同的形态，这就是生物对象的多态性。

在面向对象程序设计中，不同对象可能具有同名的函数成员。例如，使用类 A、类 B 分别定义对象 aObj 和 bObj，假设它们都有一个名为 Fun 的函数成员，但算法和功能各不相同。将调用对象函数成员 Fun 的操作做如下类比：

- **调用对象的函数成员** Fun。例如，

  ```
  aObj.Fun( );   bObj.Fun( );
  ```

  这类似于向对象 aObj、bObj 分别下达了相同的指令“Fun”。
- **执行函数** Fun。这类似于对象 aObj、bObj 各自执行指令“Fun”。
- **完成不同的功能**。这类似于对象 aObj、bObj 表现出不同的形态。

面向对象程序设计借用拟人化的说法，将调用对象的某个函数成员称为向对象发送一条**消息**，将执行函数成员完成某种程序功能称为对象**响应**该消息。不同对象接收相同的消息，但会表现出不同的行为，这就称为**对象的多态性**，或称**对象具有多态性**。从程序角度看，对象多态性就是：调用不同对象的同名函数成员，所执行的函数不同，完成的程序功能也不同。导致对象多态性的同名函数成员有以下三种不同形式：

（1）**不同类之间的同名函数成员**。类成员具有类作用域，不同类之间的函数成员可以同名，互不干扰。

（2）**类中的重载函数**。同一类中的函数成员之间可以重名，只要它们的形参个数不同，或类型不同。重载函数成员导致的多态在本质上属于重载函数多态。

（3）**派生类中的同名覆盖**。派生类中新增的函数成员可以与从基类继承来的函数成员重名，但它们不是重载函数。

对象多态性所针对的是第三种形式的同名问题，即派生类中同名的新增函数成员与基类函数成员之间所造成的多态。

### 3. 对象多态性与代码重用

为了扩展或修改基类的功能，类族中的派生类可能会定义新的函数成员来覆盖同名的基类成员。这样，同一类族中的基类及各个派生类都具有各自的同名函数成员。这些函数成员虽然函数名相同，但实现的算法会有所不同，例如钟表类族中显示时间的函数成员 Show。应用类型兼容语法规则，通过基类的引用或对象指针访问类族的任何对象，只能访问到其基类成员。如果通过基类的引用或对象指针访问时能够根据实际引用或指向的对象类型，自动调用该类同名函数成员中的新增成员，则我们称该类族中的对象具有多态性。

应用类型兼容语法规则，可以让某些处理基类对象的代码被派生类对象重用。这里“**某些**”代码的含义是：这些代码在通过基类的引用或对象指针访问派生类对象时，从代码功能上看只需要访问基类成员。例如 SetGMT 的函数代码，无论是钟表类族中的哪种对象，在设置时间都只需要调用基类函数成员 Set。

在类型兼容语法规则的基础上，面向对象程序设计进一步引入了对象多态性，其目的是让另外一些处理基类对象的代码也能够被派生类对象重用。这里“**另外一些**”代码的含义是：这些代码在通过基类的引用或对象指针访问同一类族的对象时，从代码功能上看需要根据实际引用或指向的对象类型，自动调用该类同名函数成员中的新增成员（而不是基类成员）。例如 ShowBJT 的函数代码，我们希望它在显示钟表对象时间的时候能根据实际引用或指向的对象类型，自动调用该类同名函数成员中的新增成员 Show，从而显示出带不同类标签的时间格式。C++语言以虚函数的语法形式来实现对象多态性。

## 8.5.5　虚函数

应用虚函数实现对象多态性的过程分为两步：先声明虚函数，再调用虚函数。

### 1. 声明虚函数

首先，在定义基类时使用关键字 virtual 将函数成员声明成虚函数，然后通过公有继承定义派生类，并重写虚函数成员，也就是新增一个与虚函数同名的函数成员。今后使用基类或派生类定义对象，其函数成员中只有虚函数成员才会在调用时呈现出多态性。

**举例**：定义如下的基类 A 和派生类 B。在基类 A 中将函数成员 fun1 声明成虚函数，然后通过公有继承定义派生类 B，并重写虚函数成员 fun1。为便于对照，同时在基类 A 中声明一个非虚函数 fun2，并在派生类 B 中重写该函数。

```
class A                              // 基类 A
{
public:
      virtual   void fun1( );        // 函数成员 fun1 被声明为虚函数
      void fun2( );                  // 函数成员 fun2 被声明为非虚函数
} ;
void A::fun1 ( )   {   cout << "Base class A: virtual fun1( ) called." << endl;   }
void A::fun2 ( )   {   cout << "Base class A: non-virtual fun2( ) called." << endl;   }
```

```
class B : public A                    // 派生类 B
{
public:
    virtual   void fun1( );           // 重写虚函数成员 fun1
    void fun2( );                     // 重写非虚函数成员 fun2
} ;
void B::fun1 ( )   {   cout << "Derived class B: virtual fun1( ) called." << endl;   }
void B::fun2 ( )   {   cout << "Derived class B: non-virtual fun2( ) called." << endl;   }
```

函数成员 fun1、fun2 并没有实际意义。其函数体中只有一条 cout 指令输出特定的提示信息，用于确认哪个函数被调用执行了。

声明虚函数的语法细则：

（1）只能在类声明部分声明虚函数。在类实现部分定义函数成员时不能再使用关键字 virtual。

（2）基类中声明的虚函数成员被继承到派生类后，自动成为派生类的虚函数成员。

（3）派生类可以重写虚函数成员。如果重写后的函数原型与基类虚函数成员完全一致，则该函数自动成为派生类的虚函数成员，无论声明时加不加关键字 virtual。

（4）类函数成员中的静态函数、构造函数不能是虚函数。析构函数可以是虚函数。

### 2. 调用虚函数

定义一个基类 A 的对象 aObj 和一个派生类 B 的对象 bObj，然后分别通过对象名、基类引用、基类对象指针调用函数成员 fun1 和 fun2。对比调用虚函数成员与非虚函数成员的结果，看看调用哪种函数成员会呈现多态性，以及以什么形式调用才会呈现多态性。具体代码如下：

```
A   aObj;                  // 定义一个基类对象 aObj
B   bObj;                  // 定义一个派生类对象 bObj

// 演示 1：通过对象名分别调用虚函数成员 fun1 和非虚函数成员 fun2，对比调用结果
bObj.fun1( );              // 调用结果：调用了派生类对象 bObj 的新增函数成员 fun1
bObj.fun2( );              // 调用结果：调用了派生类对象 bObj 的新增函数成员 fun2
// 结论 1：通过对象名访问派生类对象 bObj 的成员，访问的都是新增成员（同名覆盖）

// 演示 2：通过基类引用分别调用虚函数成员和非虚函数成员，对比调用结果
A   &raObj = aObj;         // 定义一个基类引用 raObj，引用基类对象 aObj
raObj.fun1( );             // 调用结果：调用了基类对象 aObj 的虚函数成员 fun1
raObj.fun2( );             // 调用结果：调用了基类对象 aObj 的非虚函数成员 fun2
A   &rbObj = bObj;         // 定义一个基类引用 rbObj，引用派生类对象 bObj
rbObj.fun1( );             // 调用结果：调用了派生类对象 bObj 的新增函数成员 fun1
rbObj.fun2( );             // 调用结果：调用了派生类对象 bObj 的基类函数成员 fun2
// 结论 2：通过基类引用访问派生类对象的虚函数成员将访问其新增成员（多态）
// 通过基类引用访问派生类对象的非虚函数成员将访问其基类成员（类型兼容规则）

// 演示 3：通过基类对象指针分别调用虚函数成员和非虚函数成员，对比调用结果
A   *paObj = &aObj;        // 定义一个基类对象指针 paObj，指向基类对象 aObj
paObj->fun1( );            // 调用结果：调用了基类对象 aObj 的虚函数成员 fun1
paObj->fun2( );            // 调用结果：调用了基类对象 aObj 的非虚函数成员 fun2
```

```
A   *pbObj = &bObj;        // 定义一个基类对象指针 pbObj，指向派生类对象 bObj
pbObj->fun1( );            // 调用结果：调用了派生类对象 bObj 的新增函数成员 fun1
pbObj->fun2( );            // 调用结果：调用了派生类对象 bObj 的基类函数成员 fun2
// 结论 3：通过基类对象指针访问派生类对象的虚函数成员将访问其新增成员（多态）
// 通过基类对象指针访问派生类对象的非虚函数成员将访问其基类成员（类型兼容规则）
```

**总结**：通过基类的引用或对象指针访问类族中对象的虚函数成员（例如 fun1），基类对象和派生类对象将分别调用各自的虚函数成员，呈现出多态性；如果访问的是非虚函数成员（例如 fun2），则访问的都是基类成员，不会呈现出多态性。

实现基类对象与派生类对象之间的多态性要满足以下三个条件：

（1）在基类中声明虚函数成员。

（2）派生类需公有继承基类，并重写虚函数成员（属于新增成员）。

（3）通过基类的引用或对象指针调用虚函数成员。

只有满足这三个条件，基类对象和派生类对象才会分别调用各自的虚函数成员，呈现出多态性。应用对象多态性，通过基类的引用或对象指针来访问派生类对象，这相当于是用基类来代表派生类。

将源程序中具有多态性的虚函数名转换成某个具体的函数存储地址，这种函数名到存储地址的转换被称为是对函数的绑定。通过基类的引用或对象指针调用虚函数成员，到底是调用基类成员还是新增成员，这在编译时还不能确定。其绑定过程需要在程序执行时才能完成。对象多态是一种执行时多态。

### 3. 重用 ShowBJT 函数

应用虚函数实现对象的多态，可以将例 8-13 中钟表类 Clock 的函数成员 Show 改为虚函数，即在其函数头前面加关键字 virtual。

```
virtual   void Show( )      // 虚函数成员：加关键字 virtual
{   cout << hour << ":" << minute << ":" << second;   }
```

这样钟表类 Clock 及其所有派生类中的函数成员 Show 都变成了虚函数，通过基类的引用或对象指针调用不同对象的显示时间函数 Show 将呈现出多态性（带不同的类标签）。利用该多态性，钟表类族中的不同对象可以共用函数 ShowBJT，显示出带不同类标签的时间格式。例如：

```
Watch   wObj;              // 定义一个派生类 Watch 的手表对象 wObj
SetGMT( wObj, 8, 30, 0 );  // 将手表对象 wObj 的时间设为 8 点 30 分（GMT 时间）
ShowBJT( wObj );           // 显示手表对象 wObj 的时间。显示结果：BJT-16:30:0（手表）
DivingWatch   dObj;        // 定义一个派生类 DivingWatch 的潜水表对象 dObj
SetGMT( dObj, 8, 30, 0 );  // 将潜水表对象 dObj 的时间设为 8 点 30 分（GMT 时间）
ShowBJT( dObj );           // 显示潜水表对象 dObj 的时间。显示结果：BJT-16:30:0（潜水表）
```

通过综合运用类型兼容语法规则和对象的多态性，最终实现了让钟表类族共用函数 SetGMT 和 ShowBJT 算法代码的目标。

总结一下，通过对象的多态性让类族共用算法代码需按以下步骤进行编程：

（1）声明虚函数成员。定义基类时首先需确定将哪些函数成员声明成虚函数。一般将

可能被派生类修改或扩充的函数成员声明成虚函数。

（2）重写虚函数。定义派生类时公有继承基类，并重写那些从基类继承来的虚函数成员。重写虚函数成员（属于新增成员）的目的是修改或扩充基类的功能。C++语言的对象多态性能够保证派生类对象会自动使用这些新功能。

（3）通过基类引用或对象指针访问对象。编写类族共用的算法代码时，需定义基类引用或对象指针来访问类族对象（无论是基类对象，还是派生类对象），然后通过基类引用或对象指针来调用对象的函数成员。访问派生类对象，调用其中的虚函数成员时将自动调用重写的虚函数(即对象多态性)，否则自动调用从基类继承来的函数成员(即类型兼容语法规则)。

**总结**：程序员在解决新的程序设计问题时可以通过继承的方法来定义派生类。重用已有的基类代码，可以有效提高新类的开发效率。综合运用基类与派生类之间的类型兼容语法规则和对象多态性，程序员可以继续重用处理基类的算法代码，进一步降低软件开发工作量。

#### 4. 虚析构函数

构造函数和析构函数是类中的特殊函数成员。构造函数不能声明成虚函数。析构函数可以声明成虚函数，析构函数无形参，无函数类型。其声明的语法形式为：

**virtual ~类名( );**

例如，可以将某个基类 A 及其派生类 B 的析构函数声明成虚函数。在基类 A 的声明部分添加如下虚析构函数：

```
virtual  ~A( )               // 虚析构函数。本例定义在声明部分，默认为内联函数
{  cout << "Class A: Destruction." << endl;  }
```

另外，在派生类 B 的声明部分添加如下虚析构函数：

```
virtual  ~B( )               // 虚析构函数。本例定义在声明部分，默认为内联函数
{  cout << "Class B: Destruction." << endl;  }
```

下面使用动态内存分配的方法分别定义一个基类对象和一个派生类对象。动态分配的对象应使用对象指针保存地址。使用结束后通过对象指针删除对象，释放内存。可以用基类的对象指针来保存派生类对象的地址，例如：

```
A  *p1 = new A;            // 动态分配一个基类对象
A  *p2 = new B;            // 动态分配一个派生类对象，使用基类的对象指针保存其地址
// 使用对象（略）
delete  p1;                // 自动调用基类析构函数，显示：Class A: Destruction.
delete  p2;                // 自动调用派生类析构函数，显示：Class B: Destruction.
                           // 再自动调用基类析构函数，显示：Class A: Destruction.
```

可以看出，p1 和 p2 都是基类的对象指针，但 p1 指向基类对象，p2 指向派生类对象。使用 delete 运算符删除对象时，将根据所指向对象的类型自动调用不同的析构函数，呈现出多态性。删除派生类对象时，先执行派生类析构函数来析构新增成员，再执行基类析构函数来析构基类成员。如果不采用虚析构函数，那么删除派生类对象时将只会调用基类析构函数。

## 8.5.6　抽象类

使用面向对象程序设计方法来设计解决某个实际问题的计算机程序，先从实际问题中提取出一个个具体的对象，并将具有共性的对象划分成类。可以继续将多个不同类中的共性部分抽象出来形成基类，编码时再从基类进行派生，还原出各个不同的类。抽象出基类可以凝练类代码，有效减少程序中的代码重复。从对象抽象出类，从类再继续抽象出基类，这是一个“自底向上，逐步抽象”的过程。越往上，类就越宽泛、越抽象。例如下面的圆形类 Circle 和长方形类 Rectangle：

```
class Circle                    // 圆形类：声明部分
{
public:
    double r;                   // 半径：数据成员
    double Area( );             // 求面积：函数成员
    double Len( );              // 求周长：函数成员
};
// 类实现部分（省略）
class Rectangle                 // 长方形类：声明部分
{
public:
    double a, b;                // 长和宽：数据成员
    double Area( );             // 求面积：函数成员
    double Len( );              // 求周长：函数成员
};
// 类实现部分（省略）
```

求面积、周长是圆形类和长方形类的共性部分，可以再抽象出如下的几何形状类 Shape：

```
class Shape                     // 形状类：声明部分
{
public:
    double Area( );             // 求面积：函数成员
    double Len( );              // 求周长：函数成员
};
```

形状类 Shape 有两个函数成员，分别是求面积函数 Area 和求周长函数 Len。仔细分析，这两个函数成员无法定义，因为形状是一个纯抽象的概念，无法计算其面积和周长。形状类就是一种抽象类，其中有两个只声明但未定义的函数成员，它们是一种纯虚函数。

本节将结合形状类来具体介绍纯虚函数和抽象类，以及它们在面向对象程序设计中的作用。

### 1. 纯虚函数

类定义中，“只声明，未定义”的函数成员被称为**纯虚函数**。纯虚函数的声明语法形式为：

**virtual　函数类型　函数名( 形参列表 ) = 0 ;**

例如，将形状类 Shape 中的两个函数成员声明成纯虚函数：

```
class Shape    // 形状类：声明部分
{
public:
    virtual   double Area( ) = 0 ;   // 求面积：纯虚函数成员
    virtual   double Len( ) = 0 ;    // 求周长：纯虚函数成员
};
```

纯虚函数是一种虚函数，具有虚函数的特性，其中最重要的一条就是虚函数成员在调用时具有多态性。

### 2. 抽象类

含有纯虚函数成员的类就是**抽象类**。例如，上述形状类 Shape 就是一个抽象类。抽象类具有如下特性：

**1）抽象类不能实例化**

不能使用抽象类定义对象（即不能实例化），因为抽象类中含有未定义的纯虚函数，其类型定义还不完整。但可以定义抽象类的引用或对象指针，所定义的引用、对象指针可以引用或指向其派生类的实例化对象。

**2）抽象类可以作为基类定义派生类**

抽象类可以作为基类定义派生类。派生类继承抽象类中除构造函数、析构函数之外的所有成员，包括纯虚函数成员。纯虚函数成员只声明了函数原型，没有定义函数体代码。因此派生类继承纯虚函数成员时，只是继承其函数原型（即函数接口）。派生类需要为纯虚函数成员编写函数体代码，这称为**实现**纯虚函数成员。派生类如果实现了所有的纯虚函数成员，那么它就变成了一个普通的类，可以实例化。只要派生类中还有一个未实现的纯虚函数成员，那么它就还是一个抽象类，不能实例化，这时它还是只能作为基类继续往下派生，直到实现所有的纯虚函数成员后才能实例化。例如，使用抽象类 Shape 可以定义派生类 Circle、Rectangle：

| 派生类 Circle | 派生类 Rectangle |
|---|---|
| class Circle : public Shape   // 圆形类 | class Rectangle : public Shape   // 长方形类 |
| { | { |
| public: | public: |
| double r;   // 半径：新增数据成员 | double a, b;   // 长和宽：数据成员 |
| Circle(double x=0)   // 构造函数 | Rectangle(double x=0, double y=0) // 构造函数 |
| {   r = x;   } | {   a = x;   b = y;   } |
| double Area( )   // 实现纯虚函数 Area | double Area( )   // 实现纯虚函数 Area |
| {   return ( 3.14*r*r );   } | {   return ( a*b );   } |
| double Len( )   // 实现纯虚函数 Len | double Len( )   // 实现纯虚函数 Len |
| {   return ( 3.14*2*r );   } | {   return ( 2*(a+b) );   } |
| }; | }; |

圆形类 Circle 和长方形类 Rectangle 继承抽象类 Shape，并且都实现了纯虚函数成员 Area 和 Len。因此它们就变成了普通的类，可以实例化。

### 3. 抽象类的应用

抽象类及其派生类构成一个类族。应用抽象类主要出于以下两个方面的考虑：统一类族的对外接口；让类族中的所有派生类对象可以共用相同的算法代码。

**1）统一类族接口**

通常，派生类继承基类是为了重用基类的代码。如果基类是抽象类，其中的纯虚函数成员只声明了函数原型，但并没有定义函数体代码。基类声明纯虚函数成员的目的不是为了重用其代码，而是为了统一类族对外的接口。在基类中声明纯虚函数成员，各派生类按照各自的功能要求实现这些纯虚函数，这样类族中所有的派生类都具有相同的接口。例如，抽象类 Shape 声明了求面积、周长的纯虚函数成员 Area 和 Len，圆形类 Circle、长方形类 Rectangle 继承并实现这两个纯虚函数。无论是圆形还是长方形，计算面积的接口被统一成 Area，计算周长的接口被统一成 Len。例如下面的演示代码：

```
Circle    cObj( 10 );                                  // 定义一个圆形类对象 cObj
Rectangle rObj( 5, 10 );                               // 定义一个长方形类对象 rObj
cout << cObj.Area( ) << "," << cObj.Len( );            // 显示圆形对象 cObj 的面积和周长
cout << rObj.Area( ) << "," << rObj.Len( );            // 显示长方形对象 rObj 的面积和周长
```

统一接口可以方便类族的使用。例如，使用类族的程序员可能会同时使用类族中的多个派生类，但只需记住一套函数成员名称。

**2）类族共用算法代码**

抽象类中定义的纯虚函数具有虚函数的特性，调用时具有多态性。在基类中声明纯虚函数成员的另一个目的是利用虚函数调用时的多态性，让类族中的所有派生类对象可以共用相同的算法代码。例如抽象类 Shape 下的所有派生类对象都可以共用函数 ShapeInfo 来显示面积和周长等形状信息：

```
void ShapeInfo( Shape *pObj )       // 显示面积和周长等形状信息
{     cout << pObj->Area( ) << "," << pObj->Len( ) << endl;      }
int main( )
{
      Circle    cObj( 10 );          // 定义一个圆形类对象 cObj
      Rectangle rObj( 5, 10 );       // 定义一个长方形类对象 rObj
      ShapeInfo( &cObj );            // 共用函数 ShapeInfo，显示圆形对象 cObj 的信息
      ShapeInfo( &rObj );            // 共用函数 ShapeInfo，显示长方形形对象 rObj 的信息
      return 0;
}
```

## 本节习题

1. 下列关于类型兼容语法规则的描述中，错误的是（　　）。
   A. 派生类的对象不能赋值给基类对象
   B. 派生类的对象可以初始化基类引用
   C. 派生类对象的地址可以赋值给基类的对象指针
   D. 应用类型兼容语法规则，实际上是将派生类对象当作基类对象来使用

2. 定义如下的基类 A 和派生类 B：

```
class A
{
public:
    virtual  void fun( )        // 函数成员 fun 被声明为虚函数
    {  cout << "A :: fun( ) called";  }
};
class B : public A
{
public:
    void fun( )                 // 重写虚函数成员 fun
    {  cout << "B :: fun( ) called";  }
};
```

执行下列代码：

```
A  *p;                          // 定义基类 A 的对象指针 p
B  bObj;                        // 定义派生类 B 的对象 bObj
p = &bObj;                      // 将基类指针 p 指向派生类对象 bObj
p->fun( );                      // 通过基类指针 p 调用虚函数成员 fun
```

通过基类指针 p 调用虚函数成员 fun，将自动调用哪个函数？（　　）

A. A::fun( )　　　　B. B::fun( )

C. 先调用 A::fun( )，再调用 B::fun( )　　　　D. 语法错误

3. 下列关于对象多态性的描述中，错误的是（　　）。

A. 通过基类引用访问派生类对象的虚函数成员，将自动调用基类继承的函数成员

B. 通过基类对象指针访问派生类对象的虚函数成员，将自动调用新增的函数成员

C. 应用对象多态性，实际上是用基类来代表派生类

D. 应用对象多态性的目的是为了让类族对象共用算法代码

4. 下列关于虚函数的描述，错误的是（　　）。

A. 声明虚函数需使用关键字 virtual

B. 基类中声明的虚函数成员被继承到派生类后仍是虚函数

C. 只有虚函数成员才会在调用时表现出多态性

D. 类中的静态函数、构造函数和析构函数都可以是虚函数

5. 下列关于纯虚函数的描述，错误的是（　　）。

A. 纯虚函数没有函数体

B. 纯虚函数会在调用时表现出多态性

C. 定义纯虚函数的目的是为了重用其算法代码

D. 含有纯虚函数成员的类被称为抽象类

6. 下列关于抽象类的描述，错误的是（　　）。

A. 不能用抽象类定义对象，即抽象类不能实例化

B. 可以用抽象类定义对象指针，指向其派生类对象

C. 可以用抽象类定义对象引用，引用其派生类对象

D. 抽象类的派生类一定是抽象类

# 8.6　关于多继承的讨论

派生类可以从多个基类继承，这就是多继承。多继承派生类存在比较复杂的成员重名问题，其具体表现形式有三种：

（1）**新增成员与基类成员重名**。在新增成员与基类成员重名的情况下访问派生类对象，所访问到的是新增成员，还是基类成员？这要由访问形式来决定。

- 如果通过派生类的对象名、引用或对象指针访问派生类对象，则访问到的是新增成员，此时新增成员覆盖同名的基类成员（**同名覆盖**）。
- 如果派生类公有继承基类，通过基类的引用或对象指针访问派生类对象的数据成员或非虚函数成员，则访问到的是基类成员。此时派生类对象被当作基类对象使用（**类型兼容语法规则**）。
- 如果基类定义虚函数成员，派生类公有继承基类并重写虚函数，则通过基类的引用或对象指针访问派生类对象的虚函数成员时，所访问到的将是重写的虚函数成员。这就是调用对象中虚函数成员时所呈现的多态性（**对象多态性**）。

（2）**多个基类之间的成员重名**。如果多个基类之间有重名的成员，同时继承这些基类会造成派生类中基类成员之间的重名。

（3）**同一基类被重复继承**。多级派生时，从同一基类派生出多个派生类，这多个派生类再被多继承到同一个下级派生类。该下级派生类将包含多份基类成员的拷贝，也就是同一基类被重复继承。

第一种重名形式，即新增成员与基类成员重名，已陆续在 8.3~8.5 节做过详细讲解。本节重点讲解另外两种重名形式，即多个基类之间的成员重名和同一基类被重复继承。

## 8.6.1　多个基类之间的成员重名

例 8-15 给出一个双继承的派生类例子。其中的派生类 B 同时继承基类 A1 和基类 A2 的成员，而这两个基类既包含不重名的成员（例如 a1、a2），也包含重名的成员（例如数据成员 a 和函数成员 fun）。

**例 8-15　一个双继承派生类的 C++示意代码**

| 基类 A1 | 基类 A2 |
| --- | --- |
| class A1 | class A2 |
| { | { |
| public: | public: |
| int a1; | int a2; |
| int a; | int a; |
| void fun( ) | void fun( ) |
| { cout << a1 << "," << a << endl; } | { cout << a2 << "," << a << endl; } |
| } ; | } ; |

双继承派生类 B

```
class   B : public A1, public A2   // 同时继承基类 A1、A2
{
public:
    // …… 不新增任何新成员，因此派生类 B 只包含从基类 A1、A2 继承的基类成员
} ;
```

继承后，派生类 B 的成员总共有 6 个：

■ 从类 A1 继承的基类成员：a1、a、fun( )

■ 从类 A2 继承的基类成员：a2、a、fun( )

■ 新增成员：无

上述 6 个成员中有两对重名成员。在访问派生类 B 的对象时该如何区分其重名成员呢？答案是在成员名之前使用作用域运算符“::”指明基类。例如：

```
B   bObj;                          // 定义派生类对象 bObj
cin >> bObj.a1 >> bObj.a2;         // 访问不重名的基类成员，直接使用成员名
cin >> bObj.A1::a >> bObj.A2::a;   // 访问重名的基类成员，需在成员名前加“基类名::”
bObj.A1::fun( );                   // 调用从类 A1 继承来的基类函数成员 fun
bObj.A2::fun( );                   // 调用从类 A2 继承来的基类函数成员 fun
```

访问上述重名的基类成员 a、fun 时，必须在成员名前加“基类名::”，否则将出现二义性，编译时会提示语法错误。例如：

```
cin >> bObj.a;                     // 语法错误：出现二义性，计算机不知道该访问哪个基类成员 a
bObj.fun( );                       // 语法错误：出现二义性，计算机不知道该调用哪个基类成员 fun
```

如果在派生类 B 中再新增一个数据成员 a：

```
class   B : public A1, public A2   // 继承基类 A1、A2
{
public:
    int   a;                       // 新增数据成员 a，与另外两个基类成员重名
} ;
```

此时又该按如何区分这三个重名成员 a 呢？答案是：访问新增成员直接使用成员名 a，访问重名的基类成员时需在成员名前加“基类名::”。例如：

```
B   bObj;                          // 定义派生类对象 bObj
cin >> bObj.a;                     // 访问新增成员 a。新增成员将覆盖同名的基类成员，即同名覆盖
cin >> bObj.A1::a >> bObj.A2::a;   // 访问重名的基类成员，需在成员名前加“基类名::”
```

## 8.6.2　重复继承

例 8-16 给出一个重复继承的例子。其中的派生类 A1、A2 分别继承基类 A，然后再定义二级派生类 B 同时继承类 A1 和 A2。

例 8-16　一个重复继承的 C++示意代码

基类 A

```
class  A
{
public:
    int  a;
    void  fun( )  {  cout << a << endl;  }
} ;
```

派生类 A1

```
class  A1 : public A   // 继承基类 A
{
public:
    // …… 不新增任何新成员
    // 派生类 A1 继承了一份基类 A 的成员
} ;
```

派生类 A2

```
class  A2 : public A   // 继承基类 A
{
public:
    // …… 不新增任何新成员
    // 派生类 A2 也继承了一份基类 A 的成员
} ;
```

二级派生类 B

```
class  B : public A1, public A2   // 同时继承类 A1、A2
{
public:
    // …… 不新增任何新成员
    // 派生类 B 同时继承类 A1、A2，继承后将包含两份完全相同的基类 A 的成员
} ;
```

首先，派生类 A1、A2 分别继承了一份基类 A 的成员，然后二级派生类 B 同时继承类 A1、A2，继承后将包含两份完全相同的基类 A 的成员，这就是重复继承。重复继承会造成派生类中的基类成员重名。继承后，派生类 B 的成员总共有 4 个：

- 从类 A1 继承的基类成员：a、fun( )
- 从类 A2 继承的基类成员：a、fun( )
- 新增成员：无

上述两对重名成员都是间接从基类 A 继承来的。定义派生类 B 的对象，在成员名之前需加作用域运算符“::”指明基类，这样才能在访问对象成员时区分重名成员。例如：

```
B  bObj;                              // 定义派生类对象 bObj
cin >> bObj.A1::a >> bObj.A2::a;      // 访问重名的基类成员，需在成员名前加“基类名::”
bObj.A1::fun( );                      // 调用从类 A1 继承来的基类函数成员 fun
bObj.A2::fun( );                      // 调用从类 A2 继承来的基类函数成员 fun
```

访问上述重名的基类成员 a、fun 时，必须在成员名前面指明**直接基类** A1 或 A2，而不能用**间接基类** A，否则也将出现二义性，编译时会提示语法错误。例如：

```
cin >> bObj.A::a;                     // 语法错误：出现二义性，计算机不知道该访问哪个基类成员 a
bObj.A::fun( );                       // 语法错误：出现二义性，计算机不知道该调用哪个基类成员 fun
```

### 8.6.3　虚基类

多级派生时，从同一基类派生出多个派生类。如果这多个派生类再被多继承到同一个下级派生类，该下级派生类将包含多份基类成员的拷贝，也就是同一基类被重复继承。重复继承时，虽然可以使用作用域运算符“::”指明基类，区分重名的基类成员，但多份同一基类的成员拷贝没有什么实际价值，反而会造成使用上的混乱，另外也浪费内存。

C++语言引入了虚基类的概念。对于重复继承时不希望保留多份成员拷贝的基类，在第一级派生时就使用关键字 **virtual** 将其声明为**虚基类**。多级派生时，即使虚基类被重复继承，其派生类也只会保留一份虚基类的成员。修改例 8-16 的代码，将基类 A 声明为虚基类。具体的声明方法是在定义其第一级派生类 A1、A2 时增加关键字 virtual，修改后的示意代码如下：

| 派生类 A1 | 派生类 A2 |
| --- | --- |
| class　A1 : **virtual** public A　// 继承虚基类 A | class　A2 : **virtual** public A　// 继承虚基类 A |
| { | { |
| public: | public: |
| 　// …… 不新增任何新成员 | 　// …… 不新增任何新成员 |
| 　// 派生类 A1 继承了一份基类 A 的成员 | 　// 派生类 A2 也继承了一份基类 A 的成员 |
| } ; | } ; |

例 8-16 中的其他代码（基类 A 和二级派生类 B 的定义代码）保持不变。派生类 A1 和派生类 A2 仍然会分别继承一份基类 A 的成员，但二级派生类 B 同时继承 A1 和 A2 却只会继承一份基类 A 的成员，因为基类 A 是虚基类。继承后，派生类 B 的成员只有两个：

- 从虚基类间接继承来的基类成员：a、fun( )
- 新增成员：无

定义派生类 B 的对象，对象只包含一个数据成员 a 和一个函数成员 fun，可直接用成员名进行访问。例如：

```
B   bObj;                // 定义派生类对象 bObj
cin >> bObj.a;           // 访问数据成员 a
bObj.fun( );             // 调用函数成员 fun
```

本节最后简单讨论一下多继承的利弊。类的继承与派生主要有两个作用：

（1）**重用类代码**。派生类继承基类，主要是重用基类的代码，使用其功能，从而提高程序开发的效率。

（2）**统一类族接口**。抽象类中包含纯虚函数成员。纯虚函数只声明函数原型，没有定义代码，没有实现任何程序功能。如果基类是抽象类，在基类中声明纯虚函数成员，其派生类按照各自的功能要求实现这些纯虚函数。这使得以该基类为根的类族中的所有派生类都具有相同的对外接口，更便于类族的使用。

多继承会造成重复继承。为解决重复继承中的多拷贝问题，C++语言又引入了虚基类。虚基类又会引出更复杂的语法。例如，如果虚基类只定义了一个带形参（无默认值）的构

造函数，那么整个继承关系中的所有直接或间接继承虚基类的派生类，都必须在构造函数的初始化列表中对虚基类的成员进行初始化。C++语言因为使用多继承，引发了一系列非常复杂的语法规则，并且复杂到难以掌握，甚至到了“语法陷阱”的程度。

后来提出的面向对象程序设计语言都放弃了多继承，例如 Java 和 C#语言。Java 和 C#语言只允许单继承，派生类只能继承一个基类，即只能重用一个基类的代码。但为了统一类族接口，它们引入了一个新的概念——**接口**（interface）。接口类似于抽象类，但接口只包含纯虚函数，不能包含数据成员。派生类可以继承多个接口，但不会造成重复继承中的多拷贝问题。因为取消了多继承，Java 和 C#语言在继承与派生方面的语法要比 C++简单一些。

C++语言经过三十余年的发展，已经积累了大量编写好的、可实现各种不同功能的类。这些类是以类库的形式提供的，例如 C++语言本身提供的标准库（类似于 C 语言的系统函数）、Microsoft 公司提供的 MFC 类库（用于开发 Windows 图形界面程序）、Intel 公司提供的 OpenCV 图像库（用于图像处理和机器视觉）等。类库相当于是已经编写好的程序零件。重用类库中的类，相当于是用现成的零件来组装程序，这样就能快速开发出功能强大的软件。第 9 章和第 10 章，我们将介绍 C++标准库和微软 MFC 类库的使用。

### 本节习题

1. 下列关于多继承的描述，错误的是（　　）。

   A. 派生类可以从多个基类继承，这就是多继承

   B. 多继承会造成从不同基类继承的成员间重名

   C. 多继承时，不同基类必须使用相同的继承方式

   D. 多继承时，派生类不能继承基类的构造和析构函数

2. 派生类从基类 A 和 B 各继承了一个数据成员 x。如需访问派生类对象 obj 中从基类 A 继承来的成员 x，下列哪种访问形式是正确的？（　　）

   A. obj.x　　B. obj.A::x　　C. obj.B::x　　D. obj.A->x

### 学习本章的要点

- 读者需学会使用组合和继承的方法来定义新类，这样可以提高类代码的开发效率。
- 读者应理解类在组合或继承时可以进行二次封装。
- 读者应从提高算法代码可重用性的角度去理解对象多态性。
- 读者只需要了解多继承的基本原理即可。多继承会导致语法陷阱。新的面向对象程序设计语言（例如 Java 和 C#）已不再支持类的多继承。

## 8.7 本章习题

1. **阅读程序**。阅读下列 C++程序。阅读后请说明程序的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
#include <math.h>
using namespace std;
class CPoint
{
private:    int x, y;
public:
        CPoint(int a, int b)    {    x = a;    y = b;    }
        int GetX( )    {    return x;    }
        int GetY( )    {    return y;    }
};
class CLine
{
private:    CPoint p1, p2;
public:
        CLine(int x1, int y1, int x2, int y2) : p1(x1, y1), p2(x2, y2)    { }
        double Length( )
        {
                int d1 = p1.GetX( ) - p2.GetX( );
                int d2 = p1.GetY( ) - p2.GetY( );
                return sqrt( d1*d1 + d2*d2 );
        }
};
int main( )
{
        CLine    line( 1, 1, 4, 5);
        cout << line.Length( ) << endl;
        return 0;
}
```

2. **阅读程序**。阅读下列 C++程序。阅读后请说明程序的功能，并对每条语句进行注释，说明其作用。

```
#include <iostream>
using namespace std;
class CBase
{
private:    int x, y;
public:
        CBase(int p1, int p2)    {    x = p1;    y = p2;    }
        virtual void Show( )    {    cout << x << ", " << y << endl;    }
};
class CDerived : public CBase
{
private:    int a, b;
public:
        CDerived(int p1, int p2, int p3, int p4) : CBase(p1, p2)
        {    a = p3;    b = p4;    }
        void Show( )    {    CBase::Show( );    cout << a << ", " << b << endl;    }
};
int main( )
```

```
{
    CBase   obj( 2, 4 );
    CBase *p = &obj;
    p->Show( );
    CDerived   obj1( 1, 3, 5, 7 );
    p = &obj1;
    p->Show( );
    return 0;
}
```

3. **编写程序**。编写计算圆环面积的 C++程序。要求先设计一个圆形类 Circle，再基于类 Circle 使用组合的方法定义一个圆环类 Ring。圆环可认为是由一大一小两个同心圆组合而成。

4. **编写程序**。编写计算圆环面积的 C++程序。要求先设计一个圆形类 Circle，再基于类 Circle 使用继承的方法定义一个圆环类 Ring。圆环可认为是在圆形基础上再增加一个描述线宽的属性，即带边框的圆形。

5. **编写程序**。编写一个演示类型兼容语法规则的 C++程序。要求先设计一个基类 A，其中包含一个函数成员 fun。再定义一个基类 A 的派生类 B，重写函数 fun（即新增一个同名函数成员 fun）。编写主函数，分别查看通过基类引用和对象指针调用派生类对象函数成员 fun 时将调用哪个函数。

6. **编写程序**。编写一个演示对象多态性的 C++程序。要求先设计一个基类 A，其中包含一个虚函数成员 fun。再定义一个基类 A 的派生类 B，重写函数 fun（即新增一个同名函数成员 fun）。编写主函数，查看通过基类引用分别访问基类对象、派生类对象成员 fun 时呈现的对象多态性，再查看通过基类对象指针分别访问基类对象、派生类对象成员时呈现的对象多态性。

# 第9章 流类库与文件I/O

程序的功能是对数据进行处理。通常，原始数据需要用户通过输入设备输入到计算机，而处理结果则需要通过输出设备反馈给用户。最常用的输入设备是键盘，被称为标准输入。最常用的输出设备是显示器，被称为标准输出。

C++语言将数据从键盘输入到某个内存变量，或将某个内存变量中的数据输出到显示器的过程看作是一种数据流动的过程。站在内存变量的角度，键盘是一种提供输入数据的数据源，显示器则是一种输出数据时的目的地。C++语言将提供输入数据的数据源称作**输入数据流**（input data stream），将输出数据时的目的地称作**输出数据流**（output data stream）。输入数据流和输出数据流统称为**输入/输出流**（I/O stream）。

C 语言通过输入/输出函数实现了数据的输入和输出，例如格式化输入函数 scanf 和输出函数 printf。C++语言则是通过输入/输出流类为程序员提供了输入或输出的功能。C++语言提供了多个输入/输出流类，可实现不同的输入/输出功能。这些输入/输出流类都是从类 ios 派生出来的，组成了一个以 ios 为基类的类族，这个类族被称为 C++语言的**流类库**。

流类库主要包含如下三组不同功能的输入/输出流类：

（1）**通用**输入/输出流类：提供通用的输入/输出（简称**标准** I/O）功能。

（2）**文件**输入/输出流类：提供文件输入/输出（简称**文件** I/O）功能。

（3）**字符串**输入/输出流类：提供字符串输入/输出（简称**字符串** I/O）功能。

C++程序员使用流类库，就是用流类库中的类定义流对象，然后访问流对象及其成员，这样就可以实现特定的输入/输出功能。流类库不属于 C++语言的主体，是其附属组成部分。使用流类库时需要用#include 指令包含相应的类声明头文件。

通过本章的学习，读者一方面可以掌握流类库的使用方法，另一方面也可以从侧面了解流类库中的类是如何设计、编写的。流类库是由全球顶尖的 C++程序员设计出来的，其中所运用的就是 C++语法知识，例如类的继承与派生、构造函数与析构函数、静态成员与常成员以及运算符重载等。通过流类库的学习，还可以帮助读者进一步深入体会前面章节所介绍的各种面向对象程序设计知识。

## 9.1 流类库

本节首先简单介绍一下输入/输出数据过程中的格式化和数据缓冲，然后再介绍流类库的组成及继承关系。

### 1. 格式化输入/输出

数据在输入/输出过程中需要进行格式转换。通常，用户通过键盘将原始数据输入到计算机，计算机再通过显示器将处理结果反馈给用户。计算机以二进制格式存储数据，而我们人类只能阅读文本格式（即字符串格式）的数据。格式转换涉及两个方面的内容。

**1）二进制数值与字符串之间的转换**

例如，一个 short 型整数 20，它在计算机内部的二进制格式是“00000000 00010100”，占 2 个字节。要想将该数据输出给人看，必须将它转换成字符串“20”，其中'2'和'0'分别是一个字符。这就是二进制数值与字符串之间的转换。

**2）多种不同的文本格式**

人类的阅读格式很灵活。例如下面的文本格式都与“20”等价：

```
"  20"              // 位数为 4 位，不足部分补空格
"20  "              // 位数为 4 位，左对齐
" +20"              // 位数为 4 位，数字前面加正负号
"0x14"              // 十六进制
```

在输出实数数值时，其格式还会涉及保留几位小数，是否采用科学表示法等。数据在输入/输出过程中，如果涉及格式转换则称为**格式化输入/输出**。

### 2. 输入/输出流中的数据缓冲区

在计算机内部，从键盘输入的数据先被存放到一个内存缓冲区（buffer），如图 9-1(a)所示。当用户按下回车键时，cin 指令再从缓冲区读取数据，经格式转换后赋值给输入变量。cout 指令在输出表达式结果时，也是先将输出数据存入一个内存缓冲区，如图 9-1(b)所示。当缓冲区满时再将其中的数据送往显示器。

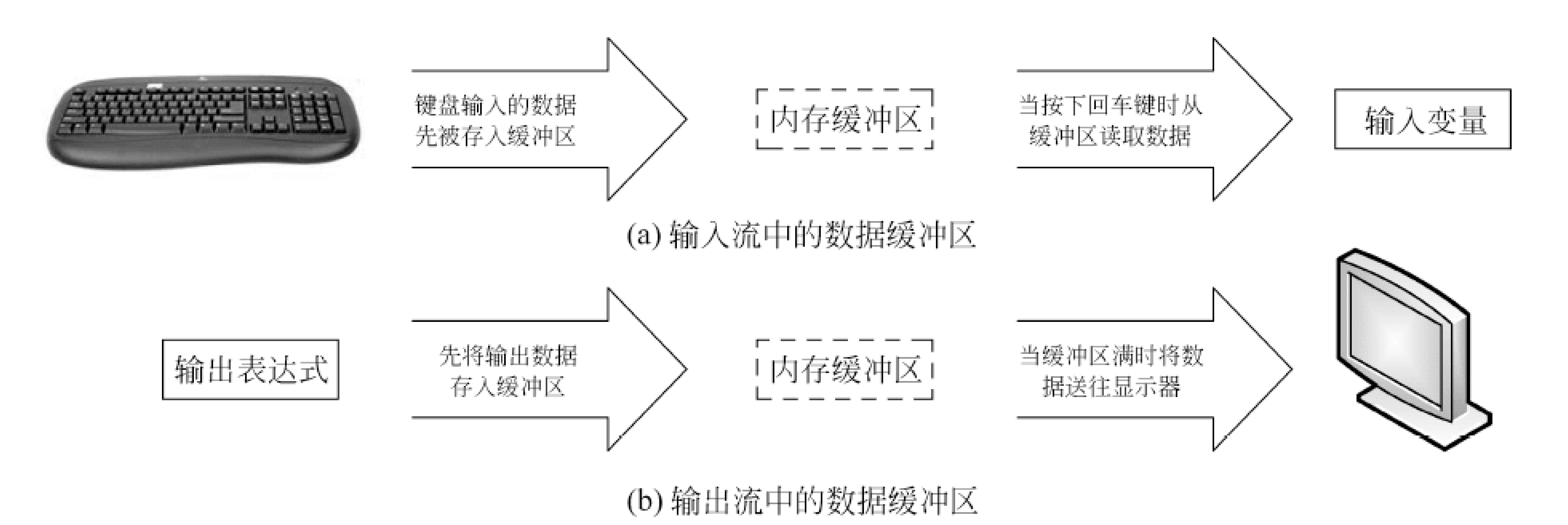



图 9-1　输入/输出流中的数据缓冲区

在流类库中，输入/输出流中的数据缓冲区被称为**流缓冲区**。流缓冲区是由流类库中一个名为 streambuf 的类来实现的。程序员不需要了解 streambuf 类的细节，但需要记住一点：数据在输入/输出过程中要在缓冲区中暂存，而不是立即输入或输出。

### 3. 流类库的基类 ios

流类库综合考虑不同的输入/输出应用，将其共性部分抽象出来形成基类 ios，其示意代码如例 9-1 所示。

**例 9-1　基类 ios 的示意代码**

```
class  ios     // 基类 ios 的声明部分
{
public:
    enum  fmtFlags {           // 枚举类型 fmtFlags：定义用作格式标记的枚举常量
        skipws     = 0x0001,   // 输入时跳过空白字符（空格、制表符、回车或换行符）
        left       = 0x0002,   // 输出时左对齐，不足部分在右侧补填充字符
        right      = 0x0004,   // 输出时右对齐，不足部分在左侧补填充字符
        internal   = 0x0008,   // 输出时在正负号或数制后面补填充字符
        dec        = 0x0010,   // 输入/输出时，将整数按十进制进行转换
        oct        = 0x0020,   // 输入/输出时，将整数按八进制进行转换
        hex        = 0x0040,   // 输入/输出时，将整数按十六进制进行转换
        showbase   = 0x0080,   // 输出时包含指示数制的基字符（例如 0 或 0x）
        showpoint = 0x0100,    // 输出浮点数时带小数点和小数 0
        uppercase  = 0x0200,   // 输出十六进制数时用大写字母
        showpos    = 0x0400,   // 输出正数时带正号
        scientific = 0x0800,   // 输出浮点数时采用科学表示法
        fixed      = 0x1000,   // 输出浮点数时不采用科学表示法，即定点格式
    };
    enum  open_mode {          // 枚举类型 open_mode：定义用作打开模式的枚举常量
        in = 0x01,  out = 0x02,  ate = 0x04,  app = 0x08,  trunc = 0x10,  binary = 0x80  };
    enum seek_dir  {           // 枚举类型 seek_dir：定义用作文件指针移动基准的枚举常量
        beg = 0,  cur = 1,  end = 2  };
    inline long flags(long _l);    // 设置输入/输出时的格式标记，内联函数
    inline long flags( ) const;    // 返回当前的格式标记，内联函数，常函数成员
    inline int width(int _i);      // 设置下一次输入/输出时的数据位数，内联函数
    inline int width( ) const;     // 返回当前的输入/输出位数，内联函数，常函数成员
    inline char fill(char _c);     // 设置输出时的填充字符，内联函数
    inline char fill( ) const;     // 返回当前的填充字符，内联函数，常函数成员
    inline int precision(int _i);  // 设置浮点数输出时的小数位数，内联函数
    inline int precision( ) const; // 返回浮点数输出时的小数位数，内联函数，常函数成员
    inline int good( ) const;      // 返回输入/输出流状态是否正常，内联函数，常函数成员
    inline int eof( ) const;       // 返回输入/输出流是否结束，内联函数，常函数成员
    locale imbue(const locale& loc);     // 设置本地语言（例如中文或英文等）
    virtual  ~ios( );              // 虚析构函数
protected:
    streambuf*  bp;                // 流缓冲区指针
    ios( );                        // 无参构造函数
private:
    ios( const ios& );             // 拷贝构造函数
    ios& operator=( const ios& );      // 重载赋值运算符
    //以下代码省略
} ;
```

基类 ios 主要包括如下成员：

（1）**数据成员**：例如指向流缓冲区的指针（代码第 36 行）。

（2）**函数成员**：例如设置或读取格式标记的函数（代码第 23~30 行）。

（3）**构造及析构函数**：其中析构函数被定义成虚析构函数。

需要重点解释的一点就是：可以将自定义数据类型放在类声明中。例如基类 ios 中定义了三个枚举类型（代码第 4~22 行），其中所定义的枚举常量会经常使用到。

### 4. 流类库的继承关系

图 9-2 给出流类库的继承关系示意图，其中的 9 个派生类可以按功能分成三组。

（1）**标准** I/O。包括通用输入流类 istream、通用输出流类 ostream 和通用输入/输出流类 iostream。

（2）**文件** I/O。包括文件输入流类 ifstream、文件输出流类 ofstream 和文件输入/输出流类 fstream。

（3）**字符串** I/O。包括字符串输入流类 istringstream、字符串输出流类 ostringstream 和字符串输入/输出流类 stringstream。

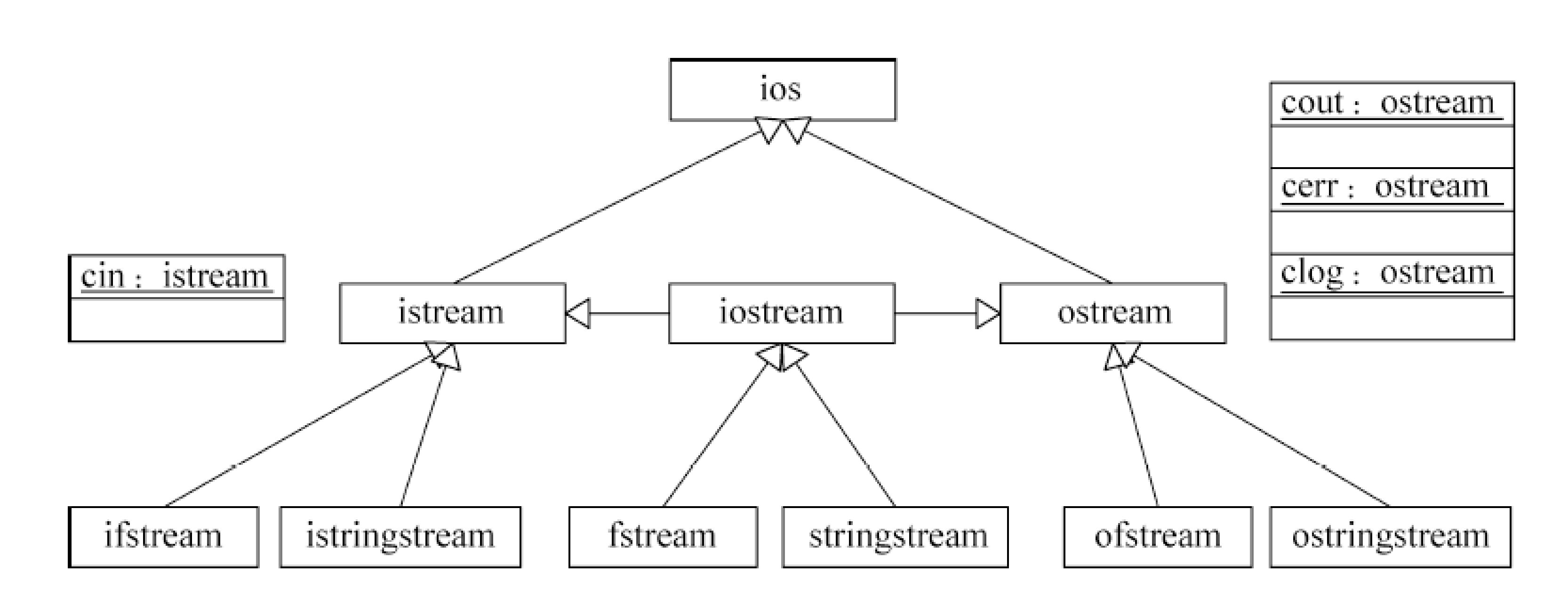



图 9-2　流类库的继承关系示意图

本节给出的流类库基类 ios 示意代码及继承关系示意图仅仅是原理性描述。不同 C++编译器所提供的流类库在用户接口上是统一的，都符合 C++标准，但其内部实现形式可能会有所区别。

图 9-2 中的 cin 是流类库中预先定义好的 istream 类对象，cout、cerr 和 clog 则是预定义的 ostream 类对象。

## 本节习题

1. 下列关于数据输入/输出的描述中，错误的是（　　）。

   A. 输入是将数据输入到内存变量中

   B. 输出是将内存中的数据输出到某个输出设备

   C. 内存中的数据是以二进制存储的

   D. 格式化输入/输出是在二进制与十进制之间进行格式转换

2. 下列关于数据输入/输出的描述中，错误的是（　　）。
   A. C 语言以系统函数的形式提供输入/输出功能
   B. C++语言通过关键字 cin、cout 以 C++语句的形式提供输入/输出功能
   C. C++语言以流类库的形式提供输入/输出功能
   D. cin 和 cout 都是流类库中预定义的流对象
3. C++流类库没有包含下列那一项内容？（　　）
   A. 标准 I/O　　B. 文件 I/O　　C. 字符串 I/O　　D. 系统函数 scanf/printf
4. 下列关于流类库的描述中，错误的是（　　）。
   A. 流类库是 C++语言的附属组成部分
   B. 流类库的作用是为 C++语言提供输入/输出功能
   C. 流类库是一个以类 ios 为基类的类族
   D. 流类库中总共定义了 3 个类

# 9.2 标准 I/O

标准 I/O 是指键盘输入，显示器输出，也被称为是控制台输入/输出。使用流类库进行输入/输出，就是用流类库中的类定义流对象，然后访问其成员，实现数据的输入和输出功能。例如，定义一个通用输入流类 istream 的对象就可以实现键盘输入的功能，定义一个通用输出流类 ostream 的对象则可以实现显示器输出的功能。

为方便程序员使用，C++流类库预先定义好了代表键盘的输入流对象 cin 和代表显示器的输出流对象 cout。其示意代码如下：

```
namespace  std                    // cin 和 cout 被定义在命名空间 std 中
{
    istream   cin;                // 键盘对象 cin
    ostream   cout;               // 显示器对象 cout
}
```

程序员可以直接使用对象 cin、cout 来输入或输出数据。例如：

```
cin >> x;                         // 输入数据流：数据从键盘 cin 流向内存变量 x
cout << x;                        // 输出数据流：数据从内存变量 x 流向显示器 cout
```

cin、cout 是类类型的对象，其所包含的函数成员和重载运算符为程序员提供了丰富的输入/输出功能。使用 cin、cout 对象需插入头文件 iostream，并声明命名空间 std，例如：

```
#include <iostream>
using namespace std;
```

## 9.2.1 通用输入流类 istream 及其对象 cin

通用输入流类 istream 公有继承基类 ios，是基类 ios 的直接派生类。例 9-2 给出其示意代码，其中 ios 被定义成虚基类。通用输入流类 istream 重载了右移运算符>>。重载后的右

移运算符>>被称为**提取运算符**。另外，通用输入流类 istream 还新增了 get、getline、read、gcount、seekg 和 tellg 等函数成员。

**例 9-2　通用输入流类 istream 的示意代码**

```
class  istream : virtual public ios          // 公有继承虚基类 ios
{
public:
     istream& operator>>(char *);            // 以下代码为重载右移运算符>>
     inline istream& operator>>(unsigned char *);
     inline istream& operator>>(signed char *);
     istream& operator>>(char &);
     inline istream& operator>>(unsigned char &);
     inline istream& operator>>(signed char &);
     istream& operator>>(short &);
     istream& operator>>(unsigned short &);
     istream& operator>>(int &);
     istream& operator>>(unsigned int &);
     istream& operator>>(long &);
     istream& operator>>(unsigned long &);
     istream& operator>>(float &);
     istream& operator>>(double &);
     istream& operator>>(long double &);

     int get( );                             // 以下代码为新增函数成员 get
     inline istream& get(unsigned char &);
     inline istream& get(   signed char &);
     inline istream& get(unsigned char *,int,char ='\n');
     inline istream& get(   signed char *,int,char ='\n');
     // 以下代码为新增函数成员 getline
     inline istream& getline(unsigned char *,int,char ='\n');
     inline istream& getline(   signed char *,int,char ='\n');
     inline istream& read(unsigned char *,int);      // 以下代码为新增函数成员 read
     inline istream& read(signed char *,int);
     int gcount( ) const;                            // 新增函数成员 gcount
     istream& seekg(long);                           // 以下代码为新增函数成员 seekg
     istream& seekg(long,ios::seek_dir);
     long tellg( );                                  // 新增函数成员 tellg
     // 以下代码省略
} ;
```

通用输入流类 istream 的提取运算符>>及其新增的函数成员 get、函数成员 getline 和函数成员 read 分别提供了不同的数据输入方法。cin 是属于通用输入流类 istream 的预定义键盘对象。下面就以键盘对象 cin 为例，重点讲解提取运算符和函数成员 get 的使用方法。函数成员 getline、read、gcount、seekg 和 tellg 等的用法将留到 9.3 节中再做具体讲解。

#### 1. 使用提取运算符>>

从键盘输入时，用户每按一次按键就相当于是向计算机输入了一个字符。键盘对象 cin 带有输入缓冲区，键盘输入的字符先被存入该缓冲区中。当用户按下回车键时，键盘对象

cin 的提取运算符再从缓冲区提取字符，经格式转换后赋值给输入变量。如果在一行输入多个数据，则数据之间通常用空格或 Tab 键隔开。

键盘对象 cin 的提取运算符是一种格式化输入方法。下面通过两个程序实例来具体演示提取运算符的使用细节。

**程序实例 1**：输入数值类型的数据

```
#include <iostream>
using namespace std;
int main( )
{
    int    x = 0;              // 定义 int 型变量 x，初始值设为 0
    double    y = 0;           // 定义 double 型变量 y，初始值设为 0
    cin >> x >> y;             // 使用键盘对象 cin 的提取运算符为变量 x 和 y 输入数据
    cout << "x=" << x << ", y=" << y << endl;        // 为验证输入结果，输出这两个变量的值
    return 0;
}
```

执行上述程序，当执行到 cin 语句时计算机暂停执行，等待用户输入数据。用户不同的数据输入方式将得到不同输入结果。例如：

■ 假设用户输入如下数据：

```
19    15.234<回车键>
```

则 cout 语句的显示结果为：

```
x=19, y=15.234
```

**结果分析**：输入结果符合预期。

■ 假设用户输入如下数据：

```
15.234    19<回车键>
```

则 cout 语句的显示结果为：

```
x=15, y=0.234
```

**结果分析**：输入结果不符合预期。提取运算符按照输入变量 x、y 的类型，将“15”转换成整数赋值给 x，将“.234”转换成浮点数赋值给 y，剩余的“19”仍留在缓冲区中。缓冲区中剩余的内容将会留给下一条 cin 语句。

■ 假设用户输入如下数据：

```
abcd<回车键>
```

则 cout 语句的显示结果为：

```
x=0, y=0
```

**结果分析**：输入结果不符合预期。如果输入数据与输入变量 x、y 的类型不一致，无法进行格式转换，则放弃输入。变量 x、y 仍保留其初始值 0。

**结论**：使用键盘对象 cin 输入数值类型数据时，用户所输入的数据应与程序中输入变量的类型、个数和顺序一致，否则会导致错误的输入结果。

**程序实例 2**：输入字符串类型的数据

```
#include <iostream>
using namespace std;
int main( )
{
    char str1[5], str2[8];          // 定义 char 型字符数组 str1 和 str2
    cin >> str1 >> str2;            // 使用键盘对象 cin 的提取运算符为数组输入数据
    cout << "str1=" << str1 << ", str2=" << str2 << endl;  // 输出这两个字符数组的内容
    return 0;
}
```

执行上述程序，当执行到 cin 语句时计算机暂停执行，等待用户输入数据。字符数组 str1 最多只能输入 4 个字符，str2 最多能输入 7 个字符，因为最后一个数组元素需要留给字符串结束符'\0'。如果用户输入的字符超过最大字符个数的限制，例如用户输入了如下数据：

```
abcdef   123456789<回车键>
```

则会导致数组越界错误。为避免输入字符串类型数据时可能出现的越界错误，可以使用键盘对象 cin 的函数成员 width 来限定输入字符的最大个数。修改上述代码，将 cin 语句改为如下形式：

```
cin.width( 5 );          // 设置下一输入项的最大字符个数为 5（含字符串结束符）
cin >> str1;             // 使用键盘对象 cin 的提取运算符输入 str1，最多提取 4 个字符
cin.width( 8 );          // 设置下一个输入项的最大字符个数为 8（含字符串结束符）
cin >> str2;             // 使用键盘对象 cin 的提取运算符输入 str2，最多提取 7 个字符
```

假设用户输入了如下数据：

```
abcdef   123456789<回车键>
```

则 cout 语句的显示结果为：

```
str1=abcd, str2=ef
```

**结果分析**：按照函数成员 width 所设置的最大字符个数，提取运算符将“abcd”赋值给 str1，“ef”赋值给 str2，剩余的“123456789”仍留在缓冲区中。缓冲区中剩余的内容将会留给下一条 cin 语句。

通用输入流类 istream 的函数成员 width 可以限定下一个输入项的最大字符个数（含字符串结束符），这样可以避免输入字符串类型数据时可能出现的数组越界错误。函数成员 width 所设置的最大字符个数仅对紧随其后的输入项有效。

### 2. 使用函数成员 get

键盘对象 cin 的函数成员 get 是一种非格式化输入方法。调用函数成员 get，可以从缓冲区中直接读出键盘输入的原始字符。函数成员 get 对所读出的字符不做任何格式转换，其中包括空白字符、Tab 键或回车键等。函数成员 get 主要有两种用法，一是每次只从缓冲区中读出一个字符，赋值给字符型变量；二是每次从缓冲区中读出多个字符（即字符串），赋值给字符型数组。

**程序实例 1：**每次从缓冲区中读出一个字符

```
#include <iostream>
using namespace std;
int main( )
{
    char ch;
    while (true)
    {
        ch = cin.get( );                              // 每次从流缓冲区中读出一个字符
        if (ch == '\n')    break;                     // 遇到回车键'\n'结束，跳出循环
        if (ch >= 'a' && ch <= 'z')    ch -= 32;      // 如果是小写字母，则转成大写字母
        else if (ch == ' ')    ch = '*';              // 如果是空格，则转成星号'*'
        cout << ch;                                   // 输出该字符
    }
    cout << endl;
    return 0;
}
```

假设用户输入了如下数据：

abc 1230<回车键>

则程序的显示结果为：

ABC*1230

通用输入流类 istream 的函数成员 get 还有另一种重载形式，也是每次从缓冲区中读出一个字符。例如：

```
ch = cin.get( );                // 从流缓冲区中读出一个字符
```

可改写成如下的重载形式：

```
cin.get( ch );                  // 重载形式：从流缓冲区中读出一个字符
```

函数成员 get 还有第三种重载形式，其功能是每次从缓冲区中读出一个字符串。

**程序实例 2：**每次从缓冲区中读出多个字符（即字符串）

```
#include <iostream>
using namespace std;
int main( )
{
```

```
    char str[5];                    // 定义字符型数组 str，有 5 个数组元素
    cin.get( str, 5 );              // 从缓冲区中读出多个字符，并自动添加字符串结束符'\0'
                                    // 最多读 5 个（含结束符），实际上最多只能读 4 个字符
    cout << str << endl;            // 输出 str，验证输入结果
    return 0;
}
```

假设用户输入了如下数据（少于 4 个字符）：

**abc**<回车键>

则 cout 语句的显示结果为：

abc

假设用户输入了如下数据（超过 4 个字符）：

abcdefg<回车键>

则 cout 语句的显示结果为：

abcd

函数成员 get 在输入字符串时需设置最大字符个数（含字符串结束符），这样可以避免可能出现的数组越界错误。

## 9.2.2　通用输出流类 ostream 及其对象 cout

通用输出流类 ostream 公有继承基类 ios，是基类 ios 的直接派生类。例 9-3 给出其示意代码，其中 ios 被定义成虚基类。通用输出流类 ostream 重载了左移运算符<<。重载后的左移运算符<<被称为**插入运算符**。另外，通用输出流类 ostream 还新增了 put、write、seekp、tellp 和 flush 等函数成员。

**例 9-3　通用输出流类 ostream 的示意代码**

```
class   ostream : virtual public ios   // 公有继承虚基类 ios
{
public:
      // 以下代码为重载左移运算符<<
      inline   ostream& operator<<(ostream& (__cdecl * _f)(ostream&));
      inline   ostream& operator<<(ios& (__cdecl * _f)(ios&));
               ostream& operator<<(const char *);
      inline   ostream& operator<<(const unsigned char *);
      inline   ostream& operator<<(const signed char *);
      inline   ostream& operator<<(char);
               ostream& operator<<(unsigned char);
      inline   ostream& operator<<(signed char);
               ostream& operator<<(short);
               ostream& operator<<(unsigned short);
```

```
              ostream& operator<<(int);
              ostream& operator<<(unsigned int);
              ostream& operator<<(long);
              ostream& operator<<(unsigned long);
      inline  ostream& operator<<(float);
              ostream& operator<<(double);
              ostream& operator<<(long double);
              ostream& operator<<(const void *);
              ostream& operator<<(streambuf*);

      inline  ostream& put(char);                    // 以下代码为新增函数成员 put
              ostream& put(unsigned char);
      inline  ostream& put(signed char);
              ostream& write(const char *,int);      // 以下代码为新增函数成员 write
      inline  ostream& write(const unsigned char *,int);
      inline  ostream& write(const signed char *,int);
              ostream& seekp(long);                  // 以下代码为新增函数成员 seekp
              ostream& seekp(long,ios::seek_dir);
              long tellp( );                         // 新增函数成员 tellp
              ostream& flush( );      // 新增函数成员 flush：立即输出，然后清空缓冲区
      // 以下代码省略
} ;
```

通用输出流类 ostream 的插入运算符及其新增的函数成员 put、函数成员 write 分别提供了不同的数据输出方法。cout 属于通用输出流类 ostream 的预定义显示器对象。下面就以显示器对象 cout 为例，重点讲解插入运算符和函数成员 put 的使用方法。函数成员 write、seekp 和 tellp 等的用法将留到 9.3 节中再做具体讲解。

### 1. 使用插入运算符<<

插入运算符在输出数据时，首先会将各种不同类型的数据统一转换成字符串格式，然后再输出。显示器对象 cout 带有输出缓冲区，转换后的字符串先被存入该缓冲区中，当缓冲区满时再将其中的字符串送往显示器。

显示器对象 cout 的插入运算符是一种格式化输出方法。使用插入运算符时可以设定数据的输出格式，其中包括输出位数、对齐方式、进制（整数）和精度（浮点数的小数位数）等。流类库提供了两种设定输出格式的方法，分别是格式标记或格式操纵符。

**格式标记**（flag）是基类 ios 中定义的一组枚举常量，用于表示不同的输出格式（参见例 9-1 中代码第 4~18 行）。设置输出格式，某些格式需使用函数成员 flags，而另外一些格式需通过专门的函数成员。例如：

```
cout.flags( ios::hex );      // 使用函数 flags 设置格式：以十六进制输出整数
cout.width( 5 );             // 使用专门的函数设置格式：将下一输出项的输出位数设定为 5 位
cout << 20;                  // 显示结果：␣␣␣14，其中“␣”表示空格，20 的十六进制是 14
```

**格式操纵符**（manipulator）是流类库中定义的一组函数。这些函数被分散定义在不同的头文件中，其示意代码如下。

```
// 头文件：<iomanip>
inline long   setiosflags(long _l);        // 设置某个格式标记，例如：setiosflags( ios::hex )
inline long   resetiosflags(long _l);      // 将某个格式标记恢复到其默认值
inline int    setw(int _w);       // 设置输出位数，不足部分补填充字符。仅对下一输出项有效
inline int    setfill(int _m);    // 设置填充字符。输出位数不足部分补填充字符
inline int    setprecision(int _p);        // 设置浮点数的输出精度（即小数位数）
// 头文件：<ios>
inline ios&   dec(ios& _strm);             // 以十进制输出整数
inline ios&   hex(ios& _strm) ;            // 以十六进制输出整数
inline ios&   oct(ios& _strm) ;            // 以八进制输出整数
// 头文件：<ostream>
inline ostream& flush(ostream& _outs) { return _outs.flush( ); }   // 立即输出并清空缓冲区
inline ostream& endl(ostream& _outs) { return _outs << '\n' << flush; }   // 换行并 flush
inline ostream& ends(ostream& _outs) { return _outs << char('\0'); }      // 插入结束符'\0'
```

与格式标记相比，格式操纵符可以与数据一起放在 cout 语句中，不需单独的函数调用语句，使用起来更为方便。例如：

```
#include<ios>                       // 使用<ios>中定义的格式操纵符 hex
#include < iomanip>                 // 使用<iomanip>中定义的格式操纵符 setw
cout << hex << setw( 5 ) << 20;     // 显示结果：␣␣␣ 14，其中“␣”表示空格
```

下面通过四个程序实例来具体演示插入运算符、格式标记或格式操纵符的使用细节。

**程序实例 1：** 设定输出位数及填充字符

本程序实例模拟显示一个商品价格清单。分别使用格式标记和格式操纵符设定数据项的输出位数，以及位数不足部分的填充字符。

**(a) 插入运算符 + 格式标记**

```
#include <iostream>
using namespace std;

int main( )
{
    char *name[ ] = { "手电筒", "电池" };
    double price[ ] = { 75.825, 4.1 };
    cout << "商品名称  单价" << endl;
    cout.fill( '#' );   // 位数不足部分补'#'
    for (int n=0; n < 2; n++)
    {
        cout.width( 8 );   cout << name[n];
        cout << " ";
        cout.width( 6 );   cout << price[n];
        cout << endl;
    }
    return 0;
}
```

**(b) 插入运算符 + 格式操纵符**

```
#include <iostream>
#include <iomanip>
using namespace std;
int main( )
{
    char *name[ ] = { "手电筒", "电池" };
    double price[ ] = { 75.825, 4.1 };
    cout << "商品名称  单价"
        << endl << setfill( '#' );
    for (int n=0; n < 2; n++)
    {
        cout << setw( 8 ) << name[n];
        cout << " ";
        cout << setw( 6 ) << price[n];
        cout << endl;
    }
    return 0;
}
```

输出数据时如果指定输出位数，则位数不足部分补填充字符（默认的填充字符是空格）。上述程序(a)使用格式标记指定商品名称按 8 位输出，单价按 6 位输出，位数不足部分补'#'。而程序(b)则使用格式操纵符做相同的设定。这两个程序的执行结果一样，都会按指定格式显示如下的内容：

```
商品名称 单价
##手电筒 75.825
####电池 ###4.1
```

如果不设定任何输出格式，则上述程序会使用默认格式显示如下的内容：

```
商品名称 单价
手电筒 75.825
电池 4.1
```

**程序实例 2**：设定对齐方式

本程序实例模拟显示与程序实例 1 相同的商品价格清单，但将价格清单的对齐方式设定为左对齐（默认的对齐方式为右对齐）。分别使用格式标记和格式操纵符来设定左对齐。

**(a) 插入运算符 ＋ 格式标记**

```
#include <iostream>
using namespace std;

int main( )
{
    char *name[ ] = { "手电筒", "电池" };
    double price[ ] = { 75.825, 4.1 };
    cout << "商品名称  单价\n";
    cout.flags( ios:: left );    // 左对齐
    for (int n=0; n < 2; n++)
    {
        cout.width( 8 );    cout << name[n];
        cout << " ";
        cout.width( 6 );    cout << price[n];
        cout << endl;
    }
    return 0;
}
```

**(b) 插入运算符 ＋ 格式操纵符**

```
#include <iostream>
#include <iomanip>
using namespace std;
int main( )
{
    char *name[ ] = { "手电筒", "电池" };
    double price[ ] = { 75.825, 4.1 };
    cout << "商品名称  单价\n"
        << setiosflags( ios::left );    // 左对齐
    for (int n=0; n < 2; n++)
    {
        cout << setw( 8 ) << name[n];
        cout << " ";
        cout << setw( 6 ) << price[n];
        cout << endl;
    }
    return 0;
}
```

上述程序(a)使用格式标记将价格清单的对齐方式设定为左对齐，而程序(b)则使用格式操纵符做相同的设定。这两个程序的执行结果一样，都会按指定格式显示如下的内容：

```
商品名称 单价
手电筒   75.825
电池     4.1
```

**程序实例 3**：设定输出整数时的进制

输出整数时默认为十进制，可改用八进制或十六进制输出整数。

| | (a) 插入运算符 + 格式标记 | (b) 插入运算符 + 格式操纵符 |
|---|---|---|
| 1 | #include <iostream> | #include <iostream> |
| 2 | using namespace std; | #include <iomanip> |
| 3 | | using namespace std; |
| 4 | int main( ) | int main( ) |
| 5 | { | { |
| 6 | int x = 20; | int x = 20; |
| 7 | cout << x << endl; // 默认十进制：20 | cout << x << endl; // 默认十进制：20 |
| 8 | cout.**flags**( **ios::hex** ); // 十六进制 | cout << **hex** << x << endl; // 显示结果：14 |
| 9 | cout << x << endl; // 显示结果：14 | |
| 10 | cout.**flags**( **ios::hex** \| **ios::showbase** ); | cout << **setiosflags**( **ios::showbase** ); // 设置标记 |
| 11 | cout << x << endl; // 显示结果：0x14 | cout << x << endl; // 显示结果：0x14 |
| 12 | cout.**flags**( **ios::oct** ); // 八进制 | |
| 13 | cout << x << endl; // 显示结果：24 | cout << **resetiosflags(ios::showbase)**; // 取消标记 |
| 14 | cout.**flags**( **ios::dec** ); // 十进制 | cout << **oct** << x << endl; // 显示结果：24 |
| 15 | cout << x << endl; // 显示结果：20 | cout << **dec** << x << endl; // 显示结果：20 |
| 16 | return 0; | return 0; |
| 17 | } | } |

可以用位或运算符“|”来组合多个格式标记（例如上述代码(a)第 10 行）。使用格式操纵符 setiosflags 可以设置格式标记（例如上述代码(b)第 10 行），而取消该格式标记则需要使用 resetiosflags（例如上述代码(b)第 13 行）。

**程序实例 4**：设定浮点数的输出格式

输出浮点数（实数）时，默认是根据其数值大小自动选择合适的输出格式。程序员可以指定使用定点表示法或科学表示法，也可以设置其输出精度（即所保留的小数位数）。

| | (a) 插入运算符 + 格式标记 | (b) 插入运算符 + 格式操纵符 |
|---|---|---|
| 1 | #include <iostream> | #include <iostream> |
| 2 | using namespace std; | #include <iomanip> |
| 3 | | using namespace std; |
| 4 | int main( ) | int main( ) |
| 5 | { | { |
| 6 | double x = 12.345678; | double x = 12.345678; |
| 7 | cout << x << endl; // 默认：12.3457 | cout << x << endl; // 默认：12.3457 |
| 8 | cout.**flags**( **ios::fixed** ); // 定点表示法 | cout << **setiosflags**( **ios::fixed** ) |
| 9 | cout.**precision**( **2** ); // 保留 2 位小数 | << **setprecision**( **2** ) |
| 10 | cout << x << endl; // 显示结果：12.35 | << x << endl; // 显示结果：12.35 |
| 11 | // 下面改用科学表示法 | // 下面改用科学表示法 |
| 12 | cout.**flags(ios::scientific)**; | cout << **resetiosflags(ios::fixed)** |
| 13 | cout.**precision**( **8** ); // 保留 8 位小数 | << **setiosflags(ios::scientific)** |
| 14 | | << **setprecision**( **8** ) |

| | |
|---|---|
| cout << x << endl; | << x << endl; |
| // 显示结果：1.23456780e+001 | // 显示结果：1.23456780e+001 |
| return 0; | return 0; |
| } | } |

### 2. 使用函数成员 put

显示器对象 cout 的函数成员 put 是一种非格式化输出方法，只能输出单个字符。例如下面的 put 函数成员的演示程序：

```
#include <iostream>
using namespace std;
int main( )
{
    char ch, str[ ] = "abcd";
    int n = 0;
    while (str[n] != '\0')          // 循环输出字符数组 str 中的字符
    {
        ch = str[n];                // 从数组 str 中取第 n 个字符（小写字母）
        cout << ch;                 // 用插入运算符输出字符 ch
        cout.put( ch-32 );          // 将字符 ch 转成大写字母，再用函数成员 put 输出
        n++;
    }
    cout << endl;
    return 0;
}
```

执行上述程序，其显示结果为：

```
aAbBcCdD
```

### 3. 函数成员 flush

显示器对象 cout 带有输出缓冲区。使用 cout 所输出的内容被先存入缓冲区，当缓冲区满时再输出到显示器。通用输出流类 ostream 新增了一个函数成员 flush。调用该函数可以立即输出缓冲区中的内容，然后清空缓冲区。例如：

```
cout << "Show Message";              // 执行该语句，“Show Message”被存入缓冲区
cout.flush( );                       // 立即输出缓冲区中的内容，然后清空缓冲区
```

还可以通过格式操纵符 flush 或 endl，实现与函数成员 flush 相同的功能。例如：

```
cout << "Show Message" << flush;     // 立即输出“Show Message”
```

或：

```
cout << "Show Message" << endl;      // 立即输出“Show Message\n”（多插入一个换行符）
```

除了键盘对象 cin 和显示器对象 cout，C++流类库还预定义了另外两个通用输出流类 ostream 的对象 clog 和 cerr。其示意代码如下：

```
namespace   std                         // cin、cout、clog 和 cerr 都被定义在命名空间 std 中
{
    istream    cin;                     // 标准输入对象 cin
    ostream   cout;                     // 标准输出对象 cout
    ostream   clog;                     // 标准日志对象 clog
    ostream   cerr;                     // 标准错误对象 cerr
}
```

对象 clog、cerr 的用法与 cout 相同，例如：

```
cout << "Show Message for user" << endl;            // 输出：Show Message for user
clog << "Show Message for operator" << endl;        // 输出：Show Message for operator
cerr << "Show Message for programmer" << endl;      // 输出：Show Message for programmer
```

对象 clog、cerr 和 cout 的区别是：通常，程序员用 cout 输出给用户看的正常提示信息；用 clog 输出给系统管理员看的运行日志信息，其中记录了程序的运行状态；用 cerr 输出给程序员自己看的错误信息，其中记录了程序运行过程中的内部错误。对象 clog 与 cout 一样，带有缓冲区。而对象 cerr 不带缓冲区，这样可以及时输出错误信息。

## 本节习题

1. 下列关于通用输入流类的描述中，错误的是（　　）。
   A. 流类库中，通用输入流类的类名为 istream
   B. 通用输入流类 istream 重载了右移运算符>>，称为提取运算符
   C. 提取运算符是一种格式化输入方法
   D. 通用输入流类 istream 只提供了提取运算符这一种输入方法
2. 执行 C++语句“char str[5];　cin >> str;”，此时在键盘输入下列哪项数据可能导致运行错误？（　　）
   A. 123　　B. abcd　　C. ABCD　　D. ABC123
3. 下列关于通用输出流类的描述中，错误的是（　　）。
   A. 流类库中，通用输出流类的类名为 ostream
   B. 通用输出流类 ostream 重载了右移运算符>>，称为插入运算符
   C. 插入运算符是一种格式化输出方法
   D. 除插入运算符外，通用输出流类 ostream 还提供了其他形式的输出方法
4. 下列哪种方法不能实现换行显示？（　　）
   A. cout << endl;　　B. cout << 'n';　　C. cout << '\n';　　D. cout << "\n";

# 9.3 文件 I/O

除了通过键盘输入、显示器输出的标准 I/O，用户也可能以文件形式向程序批量输入原始数据。例如，某个气象观测站观测员记录的每天的气温。使用 Windows 中的“记事本”程序，按如下格式将某个月的气温数据输入计算机：

```
日期  气温
1     30.2
2     28.7
…
30    25.9
```

将上述气温数据保存成一个硬盘文件，例如 temperature.txt。可以编写一个 C++程序分析气温数据，比如求月最高气温、最低气温及平均气温等。这时，C++程序需要直接从硬盘文件读取原始数据，这被称为**文件输入**。程序也可能需要将处理结果写入到硬盘文件中，这被称为**文件输出**。因为输出到显示器上的处理结果只能实时观看。程序一旦退出时，所有的显示结果都将丢失。而将处理结果保存成外存（例如硬盘、U 盘等）上的文件，这些外存文件可以长期保存，也可以复制或传输给其他人。

文件的输入/输出被简称为文件 I/O。本节将介绍文件的基本概念，并具体讲解如何利用流类库实现文件的输入/输出操作。

## 9.3.1 文件及其操作

按存储内容分，计算机中的文件可划分成两大类，一类是**程序文件**，另一类是**数据文件**。程序文件存储的是某个程序的可执行代码，数据文件存储的是某个程序所生成的结果数据。例如，在一台安装了 Windows 操作系统的计算机硬盘上，会有一个名为“notepad.exe”的文件，该文件就是保存“记事本”程序的程序文件。执行“记事本”程序，输入文字内容，然后保存到一个硬盘文件中。这个由“记事本”程序所创建的文件就是一个数据文件（扩展名为.txt）。再比如，Word 文字处理程序在安装后被存放在硬盘上的某个程序文件中，通常其文件名为“WINWORD.EXE”。由 Word 文字处理程序所创建的 Word 文档则属于数据文件（扩展名为.doc 或.docx）。

这里对文件 I/O 做一个限定：本章所说的文件 I/O，是指数据文件的输入/输出。文件 I/O 将介绍 C++程序如何从数据文件中输入数据，以及如何向数据文件输出数据。

### 1. 文件名

计算机以**文件**（file）为单位来管理存储在外存（硬盘、光盘、U 盘等）上的信息。当文件数量很多时，可以为文件建立分类**目录**（directory），将文件集中在不同目录下进行管理。目录下可以再建立**子目录**（subdirectory）。同一子目录下的文件不能重名。图 9-3 给出了 Windows 操作系统的目录结构示意图。Windows 操作系统也将目录称作“文件夹”。

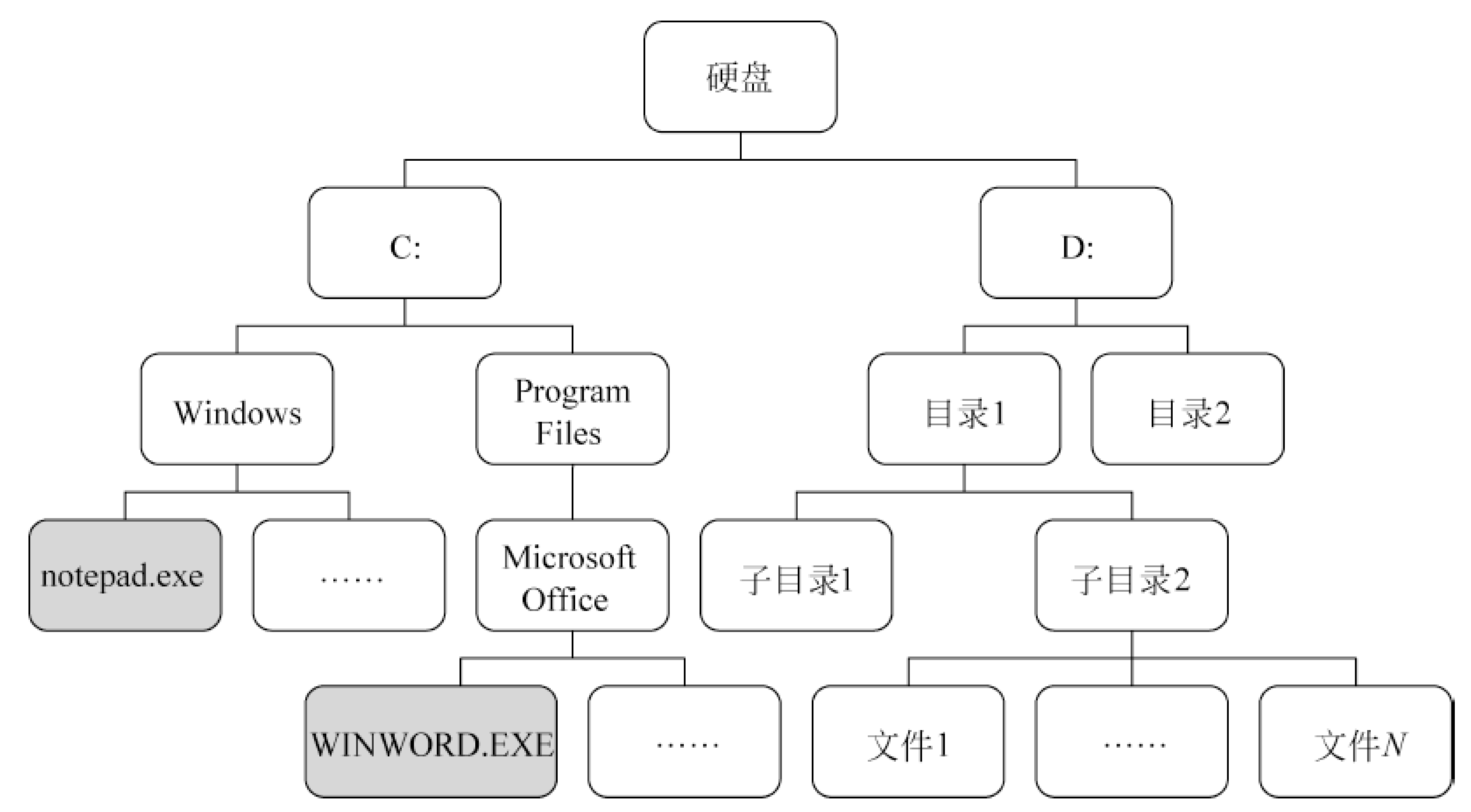



图 9-3　Windows 操作系统的目录结构示意图

文件名是文件的唯一标识。不同操作系统的文件名格式有一些区别。在 Windows 系统中，一个完整的文件名格式如下：

**盘符:\目录名\子目录名\……\文件名.扩展名**

其中，盘符、目录、子目录和文件名之间都用反斜杠“\”隔开，文件名和扩展名之间用点“.”隔开。例如，图 9-3 中“记事本”程序文件的完整文件名为：

C:\Windows\notepad.exe

在 C++源程序中，如果用字符串常量的形式来书写“记事本”程序的文件名，则应写成如下的形式：

```
"C:\\Windows\\notepad.exe"          // 字符串常量中的反斜杠应使用转义字符的形式
```

### 2. 文件格式

按存储格式分，计算机中的文件可划分成两大类，一类是**文本文件**（text file），另一类是**二进制文件**（binary file）。

#### 1）文本文件

文本文件用于存储字符类型的数据，并且主要是可见字符，例如英文字母、阿拉伯数字、标点符号，以及其他语种的文字字符（比如中文字符）等。文本文件具有如下特点：

- **存储字符编码**。文本文件存储的内容是一个字符序列。存储字符就是存储字符的编码，例如英文字母存储的是其 ASCII 编码（一个字节），中文字符存储的是其机内码（两个字节）。
- **具有换行格式**。文本换行时，存储两个控制字符 CR（ASCII 编码为 13）和 LF（ASCII 编码为 10）。注：不同操作系统可能会有所不同，例如 UNIX/Linux 操作系统的文本文件换行时只存储一个控制字符 LF。
- **通用性强**。文本文件存储的是纯文本内容，而且使用的是标准编码。文本文件不含

任何其他附加信息（例如字体、排版格式等）。阅读文本文件不需要安装特殊的软件，使用类似于“记事本”这样的常规软件就能阅读、修改。换句话说，文本文件的通用性强。

- **可用于数据交换**。文本文件通用性强，可用于数据交换。例如，一个程序的处理结果可以通过文本文件输出给人来阅读，这是程序—人之间的数据交换；一个程序的处理结果可以通过文本文件输入给另一个程序，这是程序—程序之间的数据交换；一个操作系统中程序的处理结果可以通过文本文件传送给另一个操作系统中的程序，这是操作系统—操作系统之间的数据交换。

如果是程序与人之间交换数据，那么最好使用文本文件。文本文件便于人阅读。计算机以二进制格式存储数据，因此程序在输入/输出文本文件的过程中，需要对数据在二进制格式与文本格式（即字符串格式）之间进行转换。

**2）二进制文件**

如果只是程序与程序之间交换数据，那么除了文本文件之外还可以使用二进制文件。二进制文件是直接以内存的二进制存储格式在外存上存储数据。换句话说，外存二进制文件中数据的存储格式与该数据在内存中的存储格式是一致的。计算机内存以二进制存储数据，存储时还涉及占用字节数、整数格式或浮点数格式等存储格式。不同数据类型具有不同的存储格式。使用二进制文件保存内存变量中的数据，就是直接将其内存单元的内容复制到外存的文件中，不经过任何格式转换。和文本文件相比，二进制文件具有如下特点：

- **可保存任意类型的数据**。文本文件只存储字符类型数据，任何其他类型的数据必须转换成字符串才能保存到文本文件中。而二进制文件可保存任意类型的数据。
- **存储效率高**。和文本文件相比，将内存变量中数据保存成二进制文件的效率更高。效率高具体表现在两个方面：一是无需要格式转换，保存速度快；二是保存数值型数据所占用的存储空间少。例如，保存一个 short 型整数-2100，使用二进制文件只需 2 个字节；使用文本文件需将 short 型整数-2100 转换成字符串“-2100”，保存该字符串需要 5 个字节。
- **通用性差**。二进制文件是程序员自己定义的一种私有格式。不同程序会创建不同的二进制文件。无论什么类型的数据，保存为二进制文件后都变成了二进制的 0、1 序列。二进制文件天生就是一种加密的文件。在不了解存储格式的情况下，任何程序都无法正确解释由其他程序创建的二进制文件。和文本文件相比，二进制文件的通用性差。对于二进制数据文件，通常是由哪个程序创建，就由哪个程序负责阅读、修改。
- **交换数据需遵循相同的格式标准**。为了能在不同程序间通过二进制文件交换数据，人们需要为二进制文件制定共同的格式标准。例如为了交换图像数据，人们专门制定了一些二进制图像文件的格式标准，常用的有 JPEG、BMP、GIF 和 TIFF 等。

### 3. 文件的基本操作

从文件读取数据被称为文件的输入，将数据写到文件被称为文件的输出。程序对数据文件的输入/输出操作需分三步完成。

**1）打开文件**

程序在对文件进行输入/输出操作之前，首先要打开文件。**打开**（open）文件时需指定

文件名和打开模式。**打开模式**是指程序将对文件进行何种操作，例如读数据（输入）、写数据（输出），或是两者都有（输入与输出）等。

**2）读/写数据**

文件打开以后，程序可以从文件中读数据（输入）、或向文件中写数据（输出），或是两者都有（输入与输出）。向文件读写数据可以按存储顺序从头至尾依次读写，这称为**顺序读写**。也可以指定从某个任意位置开始读写，这称为**随机读写**。

**3）关闭文件**

程序在完成对文件的输入/输出操作之后，需要关闭文件。通常情况下，数据文件只能被一个程序打开，这样可以避免读写冲突。程序在关闭文件之后，该文件才能再被其他程序打开。

C++流类库中定义了三个不同的文件输入/输出流类，它们分别是文件输出流类 ofstream、文件输入流类 ifstream 以及文件输入/输出流类 fstream。定义文件输入/输出流类的对象，通过对象调用其函数成员就可以实现文件的输入/输出功能。文件输入/输出流类的对象被称为是**文件对象**。使用文件输入/输出流类需包含相应的类声明头文件：

```
#include  <fstream>
```

## 9.3.2　文件输出流类 ofstream 及文件输出

C++流类库通过文件输出流类 ofstream 提供了文件的输出功能。该类公有继承通用输出流类 ostream，是基类 ios 的二级派生类。例 9-4 给出文件输出流类 ofstream 的示意代码。

**例 9-4　文件输出流类 ofstream 的示意代码**

```
class  ofstream : public ostream                // 公有继承通用输出类 ostream
{
public:
      ofstream( );                              // 无参构造函数
      ofstream(const char *, int =ios::out);    // 有参构造函数
      ~ofstream( );                             // 析构函数

      void open(const char *, int =ios::out);   // 打开文件
      bool is_open( ) const;                    // 检查文件是否正确打开
      void close( );                            // 关闭文件
      // 以下代码省略
} ;
```

文件输出流类 ofstream 继承了通用输出流类 ostream 的插入运算符<<及函数成员 put 和 write，另外新增了打开文件函数 open 和关闭文件函数 close 等。程序员定义文件输出流类 ofstream 的文件对象，调用其函数成员就可以实现文件输出的功能。

### 1. 输出文本文件

下面的程序例子模拟显示一个商品价格清单，同时还输出一个商品价格清单文件 price.txt。该程序例子使用显示器对象 cout（流类库预先定义的）将数据输出到显示器，使

用文件输出流类 ofstream 的文件对象 fout（程序员自己定义的）将数据输出到硬盘的文本文件 price.txt。

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
int main( )
{
    char *name[ ] = { "手电筒", "电池" };
    double price[ ] = { 75.825, 4.1 };
    int n;
    // 使用显示器对象 cout 将数据输出到显示器
    cout << "商品名称  单价\n" << setiosflags( ios::left );
    for (n = 0; n < 2; n++)
    {
        cout << setw( 8 ) << name[n];      cout << " ";
        cout << setw( 6 ) << price[n];     cout << endl;
    }
    // 使用文件输出流类 ofstream 的文件对象 fout 将数据输出到文本文件 price.txt
    ofstream   fout;                // 文件输出流类 ofstream 对象 fout 需要程序员自己定义
    fout.open( "price.txt" );       // 打开文件 price.txt，如文件不存在则创建新文件
    fout << "商品名称  单价\n" << setiosflags( ios::left );   // 标题行
    for (n = 0; n < 2; n++)
    {
        fout << setw( 8 ) << name[n];      fout << " ";
        fout << setw( 6 ) << price[n];     fout << endl;
    }
    fout.close( );                  // 关闭所打开的文件 price.txt
    return 0;
}
```

执行上述程序，显示器将显示如下的价格清单：

```
商品名称  单价
手电筒    75.825
电池      4.1
```

同时，在可执行程序所在目录还创建了一个文本文件 price.txt。使用 Windows“记事本”程序打开该文件，将能看到与上述内容相同的价格清单。

使用文件对象 fout 输出文本文件，其方法与使用显示器对象 cout 在显示器上输出数据完全一致。文件对象 fout 也是通过格式标记或格式运算符设置输出格式，通过插入运算符<<输出数据。所不同的是：

- **文件对象需程序员自己定义**。对象名也由程序员自己命名，但需符合标识符命名规则。而显示器对象 cout 是流类库预先定义好的，程序员可直接使用。
- **文件对象需要关联某个外存文件**。只有与外存文件建立起关联关系，文件对象才可以向该文件输出数据。上述文件对象 fout 通过打开文件函数 open 与文件 price.txt 建立了关联关系。也可以在定义文件对象时通过构造函数来初始化所关联的文件，

例如：

```
ofstream   fout( "price.txt" );               // 定义一个文件对象 fout，同时打开文件 price.txt
```

还可以通过文件输出流类 ofstream 的对象指针来输出文件，例如：

```
// 使用文件输出流类 ofstream 的对象指针 p 将数据输出到文本文件 price.txt
ofstream   *p;   // 定义一个文件输出流类 ofstream 的对象指针 p
p = new ofstream( "price.txt" );              // 动态分配文件对象，并打开文件 price.txt
(*p) << "商品名称  单价\n" << setiosflags( ios::left );
for (n = 0; n < 2; n++)
{
      (*p) << setw( 8 ) << name[n];       (*p) << " ";
      (*p) << setw( 6 ) << price[n];      (*p) << endl;
}
p->close( );                                  // 关闭所打开的文件 price.txt
delete p;                                     // 删除动态分配的文件对象
```

■ **使用结束后文件对象需断开与文件的关联关系**。例如，上述文件对象 fout 在完成数据输出后，使用了关闭文件函数 close 来断开与文件 price.txt 的关联关系。断开关联后的文件对象 fout 可以重复使用，可再通过打开文件函数 open 与其他文件建立新的关联关系。

### 2. 输出二进制文件

下面的程序例子模拟输出一个二进制格式的商品价格清单文件 price.dat。

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string.h>
using namespace std;
int main( )
{
      char *name[ ] = { "手电筒", "电池" };
      double price[ ] = { 75.825, 4.1 };
      int n;
      // 使用文件对象 fout 将数据输出到二进制文件 price.dat
      ofstream   fout;                             // 定义一个文件输出流类 ofstream 的文件对象 fout
      fout.open( "price.dat", ios::binary );       // 以二进制模式打开文件 price.dat
      char str[7];
      for (n = 0; n < 2; n++)
      {
            strcpy( str, name[n] );     fout.write( str, sizeof(str) );       // 输出商品名称
            fout.write( (char *)&price[n], sizeof(double) );                  // 输出价格
      }
      fout.close( );                                                          // 关闭所打开的文件 price.dat
      return 0;
}
```

使用二进制文件保存内存变量中的数据，就是直接将其内存单元的内容拷贝到外存的文件中，不做任何格式转换。与文本文件相比，输出二进制文件需注意如下的几点区别：

- **打开文件**。使用打开文件函数 open 时需增加一个表示二进制模式的实参 ios::binary。ios::binary 是基类 ios 中定义的枚举常量。
- **使用函数成员 write 输出数据**。函数 write 的原型如下：

  ```
  ostream& write( const char * buf, int bytes );
  ```

  该函数的功能是将指针变量 buf 所指向内存缓冲区的数据直接复制到二进制文件中，不做任何格式转换。形参 bytes 指定要复制的字节数。
- **相同含义的数据项应具有相同的字节数**。例如，商品名称“手电筒”和“电池”实际占用的字节数分别为 7 和 5，但在写入二进制文件时都占用了 7 个字节，即最大字节数。这么做的原因是为方便今后该二进制文件的读取，因为二进制文件没有分隔符，完全要靠字节数来分隔数据项。
- **文件大小**。二进制文件的大小可根据函数 write 所写入的字节数计算出来。分析上述程序，每行价格清单有 2 个数据项，其中商品名称占 7 字节，单价占 8 字节，共 15 个字节。2 行价格清单就需要 30 个字节。通过检查文件 price.dat 的属性，可以验证其文件大小确实是 30 个字节。
- **用“记事本”程序打开会出现“乱码”**。二进制文件需要专门的程序才能阅读或修改。例如用“记事本”程序打开二进制格式的商品价格清单文件 price.dat，将会出现如下的乱码：

  手电筒 吞烫挑 R@电池 ？ffffff@

### 9.3.3 文件输入流类 ifstream 及文件输入

C++流类库通过文件输入流类 ifstream 提供了文件的输入功能。该类公有继承通用输入流类 istream，是基类 ios 的二级派生类。例 9-5 给出文件输入流类 ifstream 的示意代码。

**例 9-5 文件输入流类 ifstream 的示意代码**

```
class  ifstream : public istream                    // 公有继承通用输入类 istream
{
public:
    ifstream( );                                    // 无参构造函数
    ifstream(const char *, int =ios::in);           // 有参构造函数
    ~ifstream( );                                   // 析构函数

    void open(const char *, int =ios::in);          // 打开文件
    bool is_open( ) const;                          // 检查文件是否正确打开
    void close( );                                  // 关闭文件
    //  以下代码省略
} ;
```

文件输入流类 ifstream 继承了通用输入流类 istream 的提取运算符>>及函数成员 get、getline 和 read，另外新增了打开文件函数 open 和关闭文件函数 close 等。程序员定义文件输入流类 ifstream 的文件对象，调用其函数成员就可以实现文件输入的功能。

### 1. 输入文本文件

使用文件输入流类 ifstream 的提取运算符>>，可以将文本文件中的数据输入给内存中的变量。而使用函数成员 getline 则可以一次从文本文件读取一行，然后将其作为一个字符串输入给内存中的字符数组。下面的程序例子演示了如何从 9.3.2 节所生成的商品价格清单文件 price.txt（文本文件）中输入数据。

```
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{
    char   name[20];
    double price;
    // 使用文件输入流类 ifstream 的文件对象 fin 从文本文件 price.txt 中输入数据
    ifstream   fin;                 // 文件输入流类 ifstream 对象 fin 需要程序员自己定义
    fin.open( "price.txt" );        // 打开文件 price.txt
    fin.getline( name, 19 );        // 读出标题行
    cout << name << endl;           // 显示所读出的标题行，显示结果：商品名称  单价
    for (int n = 0; n < 2; n++)
    {
        fin >> name >> price;       // 从文件 price.txt 中读取商品名称和单价
        cout << name << ", " << price << endl;    // 显示商品名称和单价，验证输入结果
    }
    fin.close( );                   // 关闭所打开的文件 price.txt
    return 0;
}
```

执行上述程序，显示器将显示文件 price.txt 中的价格清单，显示结果如下：

```
商品名称  单价
手电筒, 75.825
电池, 4.1
```

### 2. 输入二进制文件

下面的程序例子演示了如何从9.3.2 节所生成的二进制商品价格清单文件 price.dat 中输入数据。

```
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{
    char   name[20];
```

```
    double price;
    // 使用文件对象 fin 从二进制文件 price.dat 中输入数据
    ifstream   fin;                          // 定义一个文件输入流类 ifstream 的文件对象 fin
    fin.open( "price.dat", ios::binary );    // 以二进制模式打开文件 price.dat
    for (int n = 0; n < 2; n++)
    {
        fin.read( name, 7 );                 // 输入商品名称（7 个字节）
        fin.read( (char *)&price, 8 );       // 输入单价（8 个字节）
        cout << name << ", " << price << endl;  // 显示商品名称和单价
    }
    fin.close( );                            // 关闭所打开的文件 price.dat
    return 0;
}
```

执行上述程序，显示器将显示文件 price.dat 中的价格清单，显示结果如下：

```
手电筒, 75.825
电池, 4.1
```

从二进制文件中输入数据，就是直接将文件中的二进制内容复制到指定变量所对应的内存单元，不做任何格式转换。与文本文件相比，输入二进制文件需注意如下的几点区别：

- **打开文件**。使用打开文件函数 open 时需增加一个表示二进制模式的实参 ios::binary。ios::binary 是基类 ios 中定义的枚举常量。
- **使用函数成员 read 输入数据**。函数 read 的原型如下：

  ```
  istream& read( unsigned char * buf, int bytes );
  ```

  该函数的功能是将二进制文件中的数据直接读取到指针变量 buf 所指向的内存缓冲区，不做任何格式转换。形参 bytes 指定要读取的字节数。可以使用函数成员 gcount 获得本次实际从文件中读取的字节数，其函数原型如下：

  ```
  int gcount( ) const;
  ```

- **二进制文件完全靠字节数来分隔数据项**。要编写输入二进制文件的程序，程序员必须事先知道该文件详细的存储格式，包括每个数据项的数据类型和字节数，否则无法正确输入数据。在掌握了每个数据项的数据类型和字节数之后，程序员需在程序中定义相同类型的变量来保存所输入的数据。因此二进制文件需专门的程序才能阅读或修改，否则将出现乱码。

### 3. 检查输入文件状态

在输入/输出文件过程中需要检查文件的状态，例如文件是否已结束，文件是否已损坏等。基类 ios 定义了一些检查输入/输出流状态的函数成员，其中常用的有两个。

- **eof( )**。检查文件是否已结束。
- **good( )**。检查文件是否已损坏。

例如，下面的程序实例在输入商品价格清单文件 price.txt 的过程中，使用上述两个函数来检查输入文件的状态。

```
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{
    char   ch;
    ifstream   fin( "price.txt" );          // 定义并初始化文件输入流类 ifstream 对象 fin
    while (true )
    {
        fin.get( ch );                      // 从文件 price.txt 中每次读取一个字符
        if ( fin.eof( ) == true ||          // eof 的返回值：true-文件已结束，false-文件未结束
            fin.good( ) == false )          // good 的返回值：true-文件正常，false-文件已损坏
             break;                         // 结束文件输入
        cout.put( ch );                     // 显示所读出的字符 ch
    }
    fin.close( );                           // 关闭所打开的文件 price.txt
    return 0;
}
```

执行上述程序，显示器将显示文件 price.txt 中的价格清单，显示结果如下：

```
商品名称  单价
手电筒    75.825
电池      4.1
```

## 9.3.4　文件输入/输出流类 fstream

C++流类库还提供了既可输入，又可输出的文件输入/输出流类 fstream。该类公有继承通用输入/输出流类 iostream，而 iostream 又多继承类 istream 和 ostream，因此文件输入/输出流类 fstream 是基类 ios 的三级派生类。例 9-6 给出了类 iostream 和 fstream 的示意代码。

**例 9-6　文件输入/输出流类 fstream 的示意代码**

```
class   iostream : public istream, public ostream  // 多继承类 istream 和 ostream
{
public:
    virtual ~iostream( );                          // 虚析构函数
protected:
    iostream( );                                   // 无参构造函数
    iostream(const iostream&);                     // 有参构造函数
    //   以下代码省略
};

class   fstream : public iostream                  // 公有继承通用输入/输出类 iostream
{
public:
    fstream( );                                    // 无参构造函数
```

```
    fstream(const char *, int);            // 有参构造函数
    ~fstream( );                           // 析构函数

    void open(const char *, int);          // 打开文件
    bool is_open( ) const;                 // 检查文件是否正确打开
    void close( );                         // 关闭文件
    //  以下代码省略
};
```

文件输入/输出流类 fstream 包含如下三组成员：

（1）**从类 istream 继承的基类成员**。其中包括提取运算符>>，函数成员 get、getline、read、seekp 和 tellp 等。这些成员是从类 iostream 间接继承来的，它们提供了文件输入功能。

（2）**从类 ostream 继承的基类成员**。其中包括插入运算符<<，函数成员 put、write、seekg 和 tellg 等。这些成员也是从类 iostream 间接继承而来，它们提供了文件输出功能。

（3）**新增函数成员**。其中包括打开文件函数 open 和关闭文件函数 close 等。

程序员定义输入/输出流类 fstream 的文件对象，调用其函数成员既能输入文件，也能输出文件。文件输入/输出流类 fstream 中各成员的用法与文件输入流类 ifstream 或文件输出流类 ofstream 中对应成员的用法基本一致。其中，程序员需要重点关注两点，一是打开文件函数 open，二是可能需要随机读写文件。

### 1. 打开文件函数 open

打开文件函数 open 的原型如下：

```
void open( const char * filename, int open_mode);
```

该函数的功能是建立文件对象与某个外存文件的关联关系，俗称打开文件。其中第一个形参 filename 指定外存文件的文件名，调用时其对应的实参可以是字符串常量，也可以是字符数组。第二个形参 open_moe 指定打开模式。流类库在基类 ios 中预定义了若干个表示不同打开模式的枚举常量，如表 9-1 所示。可以用位或运算符“|”来组合多个打开模式。

**表 9-1　基类 ios 中表示不同打开模式的枚举常量**

| 枚举常量 | 所表示的打开模式 |
|---|---|
| ios :: in | 打开一个输入文件。若文件不存在，则打开失败 |
| ios :: out | 打开一个输出文件。若文件不存在则创建文件，若文件存在则清空其内容 |
| ios :: binary | 以二进制模式打开文件。后续输入/输出操作应使用 read/write 函数 |
| ios :: app | 以追加（append）模式打开输出文件。后续输出数据将追加在文件的末尾 |
| ios :: ate | 打开文件，并将文件指针移到文件末尾 |
| ios :: trunc | 打开文件，并清空其内容。若未指定 ios::in、ios::app、ios::ate，则默认为此模式 |

假设已定义如下的文件对象 fobj：

```
fstream    fobj;                       // 定义一个 fstream 类的文件对象 fobj
```

下面分别给出以不同模式打开文件 aaa.dat 的示意代码。

```
fobj.open( "aaa.dat", ios::in );                    // 以输入/文本模式打开文件 aaa.dat，默认为文本模式
fobj.open( "aaa.dat", ios::in | ios::binary );      // 以输入/二进制模式打开文件 aaa.dat
fobj.open( "aaa.dat", ios::out | ios::binary );     // 以输出/二进制模式打开文件 aaa.dat
fobj.open( "aaa.dat", ios::out | ios::app );        // 以输出/追加/文本模式打开文件 aaa.dat
fobj.open( "aaa.dat", ios::out | ios::trunc );      // 以输出/清空/文本模式打开文件 aaa.dat
```

只有成功与外存文件建立起关联关系，文件对象才能对文件进行输入/输出操作。使用函数 open 打开文件有可能不成功，例如打开输入文件而该文件不存在，或打开输出文件而硬盘已满等。文件输入/输出流类提供了函数成员 is_open 来检查打开文件是否成功。

下面的程序例子从商品价格清单文件 price.txt（文本文件）中输入数据。调用打开文件函数 open 后，再调用函数 is_open 来检查其是否成功。

```
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{
    char   name[20];
    double price;
    // 使用文件输入流类 ifstream 的文件对象 fin 从文本文件 price.txt 中输入数据
    ifstream   fin;                        // 定义一个文件输入流类 ifstream 的文件对象 fin
    fin.open( "price.txt" );               // 打开文件 price.txt
    if ( fin.is_open( ) == false )         // 函数 is_open 的返回值：true-成功，false-失败
        cout << "打开文件 price.txt 失败" << endl;              // 显示错误信息
    else   // 只有文件打开成功才能继续读取数据，否则执行后续的输入语句时将出错
    {
        fin.getline( name, 19 );     cout << name << endl;        // 读出并显示标题行
        for (int n = 0; n < 2; n++)
        {
            fin >> name >> price;      // 从文件 price.txt 中读取商品名称和单价
            cout << name << ", " << price << endl;                // 显示商品名称和单价
        }
        fin.close( );                                            // 关闭所打开的文件 price.txt
    }
    return 0;
}
```

执行上述程序，若文件 price.txt 存在则显示文件中的价格清单。若删除文件 price.txt，再次执行程序则显示如下提示信息：

```
打开文件 price.txt 失败
```

### 2. 文件的随机读写

打开文件后，文件对象与外存文件建立起关联关系。此时，文件对象内部将保存当前读写数据的位置信息。该位置信息保存在被称作**文件指针**的数据成员中。文件输入流对象包含一个**读文件指针**，文件输出流对象包含一个**写文件指针**，而文件输入/输出流对象则分

别包含一个读文件指针和一个写文件指针。

通常情况下，打开文件后文件对象的读/写指针都定位于文件头的位置。每执行一次读/写操作，读/写指针将自动后移，移到下一次读/写数据的位置。这就是文件的顺序读/写。可以调用函数成员 seekg 和 tellg 来移动或读取读文件指针的位置，调用函数成员 seekp 和 tellp 来移动或读取写文件指针的位置。程序员通过移动读/写指针，可实现对文件的随机读写。这 4 个与文件指针相关函数的原型如下：

```
istream& seekg( long bytes, ios::seek_dir origin );     // 移动读文件指针
long tellg( );                                          // 返回当前读文件指针的位置
ostream& seekp( long bytes, ios::seek_dir origin );     // 移动写文件指针
long tellp( );                                          // 返回当前写文件指针的位置
```

其中，移动文件指针的函数 seekg、seekp 都有两个形参。第一个形参 bytes 表示所移动的字节数，可正可负（正数表示向后移动，负数表示向前移动）。第二个形参 origin 表示移动的起始位置，枚举常量 ios::beg 表示从文件头开始移动，ios::cur 表示从当前位置开始移动，ios::end 表示从文件尾开始移动。

下面的程序例子从二进制商品价格清单文件 price.dat 中输入数据。

```
#include <iostream>
#include <fstream>
using namespace std;
int main( )
{
    char   name[20];
    double price;
    // 使用文件对象 fin 从二进制文件 price.dat 中输入数据
    ifstream   fin;                                 // 定义一个文件输入流类 ifstream 的文件对象 fin
    fin.open( "price.dat", ios::binary );           // 以二进制模式打开文件 price.dat
    for (int n = 0; n < 2; n++)
    {
        fin.read( name, 7 );                        // 输入商品名称（7 个字节）
        fin.read( (char *)&price, 8 );              // 输入单价（8 个字节）
        cout << name << ", " << price << endl;      // 显示商品名称和单价
    }
    fin.seekg( -15, ios::end );        // 从文件尾向前（即往回）移动一行（一行有 15 个字节）
    fin.read( name, 7 );     fin.read( (char *)&price, 8 );         // 重读最后一行数据
    cout << name << ", " << price << endl;          // 显示所读出的商品名称和单价
    fin.close( );                                   // 关闭所打开的文件“price.dat”
}
```

执行上述程序，显示器将显示如下的价格清单：

```
手电筒, 75.825
电池, 4.1
电池, 4.1
```

## 本节习题

1. 下列关于文本文件的描述中，错误的是（　　）。
   A. 文本文件所存储的内容是一个字符序列
   B. 文本文件存储的是纯文本内容，而且使用的是标准编码
   C. 文本文件便于人的阅读
   D. 文本文件不能用于程序与程序之间的数据交换
2. 下列关于二进制文件的描述中，错误的是（　　）。
   A. 二进制文件以内存的二进制存储格式在外存上存储数据
   B. 将内存中二进制数据保存到二进制文件时，需要进行格式转换
   C. 和文本文件相比，二进制文件的读写速度快
   D. 和文本文件相比，二进制文件的通用性差
3. 打开一个二进制输出文件 test.dat，下列语句中错误的是（　　）。
   A. ofstream fout;　fout.open( "test.dat" );
   B. ofstream fout;　fout.open( "test.dat", ios::binary );
   C. ofstream fout( "test.dat", ios::binary );
   D. ofstream *p = new ofstream( "test.dat", ios::binary );
4. 文件输入/输出流类 fstream 不包含下列哪个函数成员？（　　）
   A. read　　B. write　　C. close　　D. delete

# 9.4　string 类及字符串 I/O

C 语言使用字符数组存储字符串数据，并在系统函数中提供了一组常用的字符串处理函数，例如 strlen、strcpy 和 strcat 等。C++语言保留了字符数组和这些字符串处理函数，另外还提供了一个新的字符串类 string。string **类**封装了字符数组和字符串处理函数，可提供更强的字符串处理功能，使用上也更加方便。

string 类的对象可以存储字符串数据。可以把 string 类对象当作数据源，将其中的字符串输入给其他变量，这时 string 类对象就是一个输入数据流。也可以把 string 类对象当作输出目的地，将其他变量中的数据输出到 string 类对象，这时 string 类对象就是一个输出数据流。在 string 类对象和其他变量之间进行数据的输入/输出，这就是字符串 I/O。

## 9.4.1　字符串类 string

字符串类string是C++语言预先定义好的类，其中封装了字符数组和字符串处理函数。程序员可以直接使用字符串类 string 定义**字符串对象**。访问字符串对象的成员，就可以实现特定的字符串处理功能。使用字符串类 string 需引入其类声明文件：

```
#include <string>
```

### 1. 字符串对象的初始化

string 类定义了多个重载的构造函数，可提供不同的对象初始化方法。下面的程序例子演示了几种字符串对象常用的初始化方法。

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
    string  str;                        // 定义字符串对象 str，未初始化
    string  str1( "Hello, world" );     // 定义字符串对象 str1，初始化方法 1
    string  str2 = "Hello, world";      // 定义字符串对象 str2，初始化方法 2
    string  str3( str2 );   // 定义字符串对象 str3，初始化方法 3（通过拷贝构造函数）

    char  cArray[ ] = "Hello, world";   // 定义字符数组 cArray
    string  str4( cArray );             // 定义字符串对象 str4，初始化方法 4
    //  以下代码省略
}
```

### 2. 字符串对象的输入与输出

可直接输入或输出字符串对象中的内容，例如：

```
    string  str;
    cin >> str;                         // 从键盘为字符串对象 str 输入内容
    cout << str << endl;                // 显示字符串对象 str 中的内容
```

### 3. string 类的函数成员

下面的程序代码演示了几种 string 类常用的函数成员：

```
string  str( "abcdefg" );
cout << str.length( ) << endl;          // 函数 length 返回字符串长度，显示结果：7
cout << str.find( "cd" ) << endl;       // 函数 find 查找子串“cd”的位置，显示结果：2
cout << str.substr( 2, 4 ) << endl;     // 函数 substr 取出一个子串，显示结果：cdef
str.append( "123" );                    // 函数 append 在串尾追加一个字符串“123”
cout << str << endl;                    // 显示结果：abcdefg123
```

### 4. string 类重载的运算符

string 类重载了某些运算符（见表 9-2。假设已定义字符串对象“string str( "abcd" );”）。重载运算符可以方便字符串之间的运算，例如赋值运算和关系运算等。

表 9-2　string 类重载的运算符

| 重载运算符 | 举　例 | 运算说明 |
|---|---|---|
| + | cout << str + "123"; | 连接字符串。显示结果：abcd123 |
| = | string str1;　str1 = str;　cout << str1; | 对象赋值。显示结果：abcd |

续表

| 重载运算符 | 举　例 | 运算说明 |
|---|---|---|
| += | str += "123";　cout << str; | 追加字符串。显示结果：abcd123 |
| == | str == "abcd" | 关系运算：等于。比较两个字符串是否相等 |
| != | str != "abcd" | 关系运算：不等于 |
| < | str < "abce" | 关系运算：小于。依次比较字符的 ASCII 码值 |
| <= | str <= "abce" | 关系运算：小于等于 |
| > | str > "abce" | 关系运算：大于 |
| >= | str >= "abce" | 关系运算：大于等于 |
| [ ] | cout << str[0]; | 下标运算：按下标访问字符。显示结果：a |

## 9.4.2　字符串 I/O

可以把字符串对象当作输入/输出数据流，在字符串对象和其他变量之间进行数据的格式化输入/输出，这就是字符串 I/O，其功能类似于 C 语言中的 sscanf 和 sprintf 函数。C++流类库通过字符串输入/输出流类提供了字符串 I/O 的功能。字符串输入/输出流类共有三个，它们分别是字符串输入流类 istringstream、字符串输出流类 ostringstream 以及字符串输入/输出流类 stringstream。

定义字符串输入/输出流类的对象，通过对象调用其函数成员就可以实现字符串 I/O 的功能。字符串输入/输出流类的对象被称为是**串流对象**。串流对象中保存的是字符串，需按照文本格式进行输入/输出。例如，使用提取运算符>>进行输入，使用插入运算符<<进行输出。使用字符串输入/输出流类需包含相应的类声明头文件：

```
#include   <sstream>
```

### 1. 字符串输入流类 istringstream

下面的程序例子演示了字符串输入流类 istringstream 的使用方法。

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int main( )
{
      string   str ( "10 25.76" );               // 定义字符串对象 str
      istringstream   strin( str );              // 定义 istringstream 类对象 strin，用 str 初始化

      int   x;
      double   y;
      strin >> x >> y;                           // 从串流对象 strin 中为变量 x、y 输入数据

      cout << "x=" << x << ", y=" << y << endl;  // 显示结果：x=10, y=25.76
      return 0;
}
```

### 2. 字符串输出流类 ostringstream

下面的程序例子演示了字符串输出流类 ostringstream 的使用方法。

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int main( )
{
    int    x = 10;
    double    y = 25.76;
    ostringstream    strout;                    // 定义 ostringstream 类对象 strout

    strout << "x=" << x << ", y=" << y;         // 向串流对象 strout 中输出数据
    string    str = strout.str( );              // 将串流对象 strout 中的字符串赋值给字符串对象 str
    cout << str << endl;                        // 显示结果：x=10, y=25.76
    return 0;
}
```

## 本节习题

1. 下列定义字符串类 string 对象的语句中，错误的是（　　）。

   A. string str;　　　　B. string str( "Hello, world" );

   C. string str = "Hello, world";　　　　D. string str = 'Hello, world';

2. 字符串类 string 重载了下列哪个运算符？（　　）

   A. +　　　　B. −　　　　C. *　　　　D. /

3. 字符串输入/输出流类不包含下列哪个类？（　　）

   A. string　　　　B. istringstream　　　　C. ostringstream　　　　D. stringstream

4. 执行下列语句：

```
istringstream    strin( "3 8.5" );
double    x = 0, y = 0;
strin >> x >> y;
```

   执行后变量 x 和 y 的值分别为（　　）。

   A. 0, 0　　　　B. 3, 8.5　　　　C. 8.5, 3　　　　C. 3, 8

# 9.5 基于 Unicode 编码的流类库

以 ios 为基类的流类库只能对基于 ANSI 编码的字符数据进行输入/输出。为了适应 Unicode 编码，C++流类库另外定义了一套基于 Unicode 编码的流类库。该流类库的风格与基于 ANSI 编码的流类库非常相似，很容易掌握。

基于 Unicode 编码的流类库以 wios 为基类，也派生出 9 个派生类。这 9 个派生类按照

功能可分为三组：

（1）**标准 I/O**。包括通用输入流类 wistream、通用输出流类 wostream 和通用输入/输出流类 wiostream。

（2）**文件 I/O**。包括文件输入流类 wifstream、文件输出流类 wofstream 和文件输入/输出流类 wfstream。

（3）**字符串 I/O**。包括字符串输入流类 wistringstream、字符串输出流类 wostringstream 和字符串输入/输出流类 wstringstream。

上述各类与 ANSI 编码流类库中的类一一对应，使用方法也类似，只是类名被冠以前缀"w"，表示宽字符。为方便程序员使用，C++流类库也预先定义了宽字符的键盘对象 wcin、显示器对象 wcout 以及另外两个宽字符对象 wclog 和 wcerr。其示意代码如下：

```
namespace   std   // wcin、wcout、wclog 和 wcerr 都被定义在命名空间 std 中
{
    wistream    wcin;           // 宽字符的标准输入对象 wcin
    wostream    wcout;          // 宽字符的标准输出对象 wcout
    wostream    wclog;          // 宽字符的标准日志对象 wclog
    wostream    wcerr;          // 宽字符的标准错误对象 wcerr
}
```

下面的程序例子简单演示了 wcout 的用法：

```
#include <iostream>
#include <iomanip>
#include <string>
#include <locale>
using namespace std;
int main( )
{
    wstring   name[ ] = {   L"手电筒", L"电池"   };        // wstring：宽字符串类
    double price[ ] = {   75.825, 4.1   };

    wcout.imbue( locale("chs") );                  // 将语言设置为简体中文 chs
    wcout << L"商品名称  单价\n";                  // 前缀 L 表示宽字符串常量
    for (int n=0; n < 2; n++)
        wcout << name[n] << L" " << price[n] << endl;
    return 0;
}
```

执行上述程序，将显示如下的内容：

```
商品名称  单价
手电筒  75.825
电池  4.1
```

注：VC 6.0 的 Unicode 编码流类库还不够完善。编写基于 Unicode 编码的 C++程序，请使用微软后续推出的新版 Microsoft Visual Studio 系列集成开发环境。

### 本节习题

1. 下列哪项内容与 Unicode 编码无关？（　　）

   A. wchar_t　　B. wstring　　C. wios　　D. char

2. 基于 Unicode 编码的流类库中，预定义的通用输入流对象是（　　）。

   A. wcin　　B. wcout　　C. wclog　　D. wcerr

### 学习本章的要点

- 读者应理解之前所用的 cin、cout 指令实际上分别是通用输入/输出流类的对象。
- 通过本章学习，读者可以从侧面了解全球顶尖的 C++程序员是如何来设计和编写类的，这样可以帮助读者进一步深入体会前面所学习的各种面向对象程序设计知识。
- 读者应重点学习如何进行文件读写操作，大部分程序都需要使用文件来保存数据。

## 9.6 本章习题

1. **编写程序**。已有定义变量语句"char str[10] = "Hello"; int x = 1234;"，编写 C++程序将变量 str 和 x 中的数据写入文本文件 test.txt。用 Windows 的记事本程序查看该文件。再编写另一个 C++程序从上述文本文件中读出数据并在显示器上显示出来。

2. **编写程序**。已有定义变量语句"char str[10] = "Hello"; int x = 1234;"，编写 C++程序将变量 str 和 x 中的数据写入二进制文件 test.dat。用 Windows 的记事本程序查看该文件。再编写另一个 C++程序从上述二进制文件中读出数据并在显示器上显示出来。

3. **编写程序**。编写一个实现文件复制的 C++程序。提示：使用二进制方式打开文件。

# 第10章 C++标准库

为了方便程序员，C++语言以**库**(library)的形式为程序员提供了很多常用的函数和类。例如本书第 6 章所介绍的系统函数以及第 9 章所学习的输入/输出流类和字符串类等，它们都是 C++语言以库的形式提供给程序员使用的。

C++语言全盘继承了 C 语言的**标准 C 库**（standard C library），其中包括非常丰富的**系统函数**，例如输入/输出函数、数学函数、字符串处理函数和动态内存分配函数等。C++语言另外又增加了一些新的库。新库中包含一些新增的系统函数，但更多的是为面向对象程序设计方法所提供的**系统类库**，这些新库被统称为 C++**标准库**（C++ standard library）。

系统类库包含一组预先定义好的类。程序员可以直接使用这些类来定义对象，也可以通过组合或继承来定义新的类。系统类库极大地扩展了 C++语言的功能，使得程序员可以在更高的起点上开发程序。程序员在使用系统类库之前，需阅读编译系统提供的手册，学习各预定义类的功能及使用方法。使用系统类库时，需用#include 指令包含其相应的类声明头文件。

C++语言的**模板**（template）技术包括**函数模板**和**类模板**。模板技术是一种代码重用技术，函数和类是 C++语言中两种主要的重用代码形式。通过模板技术可以进一步提高函数代码和类代码的可重用性。采用模板技术，首先由程序员定义代码模板，再由编译器按照模板自动生成不同的代码**实例**（instance）。在这里，代码模板就是一种被重用的代码。代码模板可以使源代码更加凝练。C++标准库在编写时就采用了模板技术，因此标准库能以较少的代码量提供很强大的功能。

本章将首先介绍函数模板和类模板，然后再具体讲解 C++标准库及其主要应用。

## 10.1 函数模板

**函数模板**的基本原理是通过数据类型的参数化，将一组算法相同但所处理数据类型不同的重载函数凝练成一个函数模板。编译时，再由编译器按照函数模板自动生成针对不同数据类型的重载函数定义代码。

例如，下面三个求最大值的重载函数 Max：

```
int      Max( int x, int y )          {  return (x>y ? x:y);  }   // 求 2 个整数的最大值
double   Max( double x, double y )    {  return (x>y ? x:y);  }   // 求 2 个实数的最大值
```

```
char      Max( char x, char y )        {  return (x>y ? x:y);  }   // 求 2 个字符的最大值
```

上述三个重载函数的算法相同，所不同的是数据类型。可以将其中的数据类型 int、double 和 char 参数化，形成一个类型参数 T，然后把上述三个函数合并成一个如下的函数模板：

```
template   <typename T>   // 定义函数模板 Max
T   Max( T x, T y )
{
      return ( x>y ? x : y );
}
```

### 10.1.1 函数模板的定义与使用

**C++语法：定义函数模板**

```
template   <类型参数列表>
函数类型   函数名( 形式参数列表 )
{
      函数体
}
```

语法说明：

- 定义函数模板以关键字 **template** 开头。
- **类型参数列表**可声明一个或多个类型参数，每个类型参数以“typename **类型参数名**”或“class **类型参数名**”的形式声明，类型参数之间用“,”隔开。类型参数名需符合标识符的命名规则。
- 函数模板定义的其余部分，包括函数类型、函数名、形式参数列表以及函数体，它们和普通函数的定义形式没有什么区别。唯一不同的是，类型参数将会作为一种新的数据类型出现在函数模板定义中。
- 使用 typename 或 class 所声明的类型参数可视为一种新的数据类型，可以用它来定义函数类型（即返回值的类型），也可以用来定义形参或在函数体中定义局部变量。类型参数是表示数据类型的参数，调用时可被替换成任意一种实际数据类型，例如某种基本数据类型、自定义数据类型或类类型等。类型参数也可理解成是一种通用数据类型。
- 函数模板定义中的前两行被称为**函数模板头**。

定义好的函数模板可像普通函数一样被调用。例 10-1 给出一个完整的求最大值函数模板 Max 的 C++演示程序。

**例 10-1 一个完整的求最大值函数模板 Max 的 C++演示程序**

```
#include <iostream>
using namespace std;

template   <typename T>   // 定义函数模板 Max，声明一个类型参数 T
T   Max( T x, T y )        // 使用类型参数 T 来定义函数的类型、形参 x 和 y 的类型
{
```

```
7        return ( x>y ? x : y );
8    }
9
10   int main( )
11   {
12       cout << Max( 5, 10 ) << endl;   // 调用函数模板 Max 求 2 个整数的最大值，显示：10
13       cout << Max( 5.2, 10.2 ) << endl;   // 调用函数模板 Max 求 2 个实数的最大值，显示：10.2
14       return 0;
15   }
```

例 10-1 程序的代码说明如下。

- **函数模板的定义**。函数模板 Max 中声明了一个类型参数 T（代码第 4 行）。T 是一种类型参数，可以用来定义函数类型，也可用来定义形参（代码第 5 行）。在编写函数模板的程序员看来，使用 typename 或 class 所声明的类型参数就像是一种新的数据类型，可以用它来定义函数类型（即返回值的类型），也可以用来定义形参或局部变量。
- **函数模板的调用**。定义好的函数模板可像普通函数一样被调用。在调用者看来，函数模板中的类型参数就像是一种通用数据类型。调用函数模板 Max 时，实参的数据类型可以是 int 型（代码第 12 行），也可以是 double 型（代码第 13 行）。

函数模板为什么能像普通函数一样调用呢，其中类型参数的作用又是什么呢？回答这两个问题首先需要了解函数模板的编译原理。

## 10.1.2　函数模板的编译原理

例 10-1 的函数模板 Max 声明了一个类型参数 T。T 是表示数据类型的参数，可指代任意一种实际数据类型。使用 C++编译器编译例 10-1 的 C++源程序，当编译到函数模板调用语句时，编译器将根据实参类型来推导类型参数 T 所指代的数据类型。

- 编译例 10-1 中代码第 12 行的函数模板调用语句：

```
cout << Max( 5, 10 ) << endl;
```

根据实参 5 和 10 的数据类型，编译器可推导出类型参数 T 所指代的数据类型应当是 int 型。当类型参数 T 确定为 int 型之后，编译器将按照函数模板 Max 自动生成如下的函数定义代码：

```
int   Max( int x, int y )   // 用数据类型 int 取代模板中的类型参数 T
{
     return ( x>y ? x : y );
}
```

然后编译器将对函数模板 Max 的调用改为对上述 int 型 Max 函数的调用。

- 编译例 10-1 中代码第 13 行的函数模板调用语句：

```
cout << Max( 5.2, 10.2 ) << endl;
```

根据实参 5.2 和 10.2 的数据类型，编译器可推导出类型参数 T 所指代的数据类型应当是 double 型。当类型参数 T 确定为 double 型之后，编译器将按照函数模板 Max 自动生成如下的函数定义代码：

```
double   Max( double x, double y )   // 用数据类型 double 取代模板中的类型参数 T
{
       return ( x>y ? x : y );
}
```

然后编译器将对函数模板 Max 的调用改为对上述 double 型 Max 函数的调用。double 型 Max 函数与 int 型 Max 函数构成了重载函数。

综上所述，函数模板的**编译原理**是：函数模板是具有类型参数的函数。类型参数是表示数据类型的参数，可指代任意一种实际数据类型。编译器在编译到函数模板调用语句时，根据位置对应关系从实参数据类型推导出类型参数所指代的数据类型，然后按照函数模板自动生成一个该类型的函数定义代码。不同类型实参的函数模板调用语句将生成不同类型的重载函数。函数模板将数据类型参数化，调用时会呈现出**参数多态性**。

按照函数模板的编译原理，例 10-1 所示的 C++程序与如下的程序代码等价。

```
#include <iostream>
using namespace std;
int   Max( int x, int y )                      // 重载函数 1：int 型 Max 函数
{   return ( x>y ? x : y );   }
double   Max( double x, double y )             // 重载函数 2：double 型 Max 函数
{   return ( x>y ? x : y );   }
int main( )
{
       cout << Max( 5, 10 ) << endl;           // 调用 int 型 Max 函数求 2 个整数的最大值
       cout << Max( 5.2, 10.2 ) << endl;       // 调用 double 型 Max 函数求 2 个实数的最大值
       return 0;
}
```

可以看出，函数模板可以凝练源代码，但编译后的机器语言代码并不会减少。与重载函数相比，使用函数模板可以减少程序员的编码工作量，但最终所编译出的可执行代码是完全一样的。例 10-2 再给出一个演示函数模板语法的 C++程序。

**例 10-2　一个演示函数模板语法的 C++程序**

| (a) 使用函数模板的程序 | (b) 等价的重载函数程序 |
| --- | --- |
| `#include <iostream>` | `#include <iostream>` |
| `using namespace std;` | `using namespace std;` |
| | |
| `template <typename T>   // 函数模板 fun` | `int   fun( int x, int y )   // 重载函数：int 型` |
| `T   fun( T x, int y )` | `{` |
| `{` | `    int   z;` |
| `    T   z;   // 定义 T 类型的变量 z` | `    z = (int)( x+y );` |
| `    z = (T)( x+y ); // 将x+y的结果转换成T类型` | `    return   z;` |
| `    return   z;` | `}` |
| `}` | `double   fun( double x, int y )   // 重载：double 型` |

| | |
|---|---|
| | { |
| | **double** z; |
| | z = (**double**)(x+y); |
| | return z; |
| | } |
| | |
| int main ( ) | int main ( ) |
| { | { |
| cout << **fun**( **10**, 5 ) << endl; // 显示：15 | cout << **fun**( **10**, 5 ) << endl; // 显示：15 |
| cout << **fun**( **10.2**, 5 ) << endl; // 显示：15.2 | cout << **fun**( **10.2**, 5 ) << endl; // 显示：15.2 |
| return 0; | return 0; |
| } | } |

例 10-2 中程序(a)和程序(b)所实现的功能完全相同。对比源程序的代码量可以看出，使用函数模板可以有效降低程序员的编码工作量。

### 10.1.3　函数模板的声明

与普通函数一样，函数模板也应当遵守“先定义，后调用”的原则。如果函数模板定义在后，或定义在其他程序文件中，则应“先声明，后调用”。声明函数模板的语法形式为：

```
template <类型参数列表>
函数类型  函数名(形式参数列表) ;          // 即：函数模板头再加分号“;”
```

或

```
template <类型参数列表>   函数类型  函数名(形式参数列表) ;  // 写在一行
```

例如，如果将例 10-2(a)中函数模板 fun 的定义放在主函数 main 的后面，则应按如下方式先声明，后调用。

```
#include <iostream>
using namespace std;
template    <typename T>
T    fun( T x, int y ) ;                // 函数模板 fun 的声明
int main ( )
{   // 代码省略    }
template    <typename T>                // 函数模板 fun 的定义
T    fun( T x, int y )
{   //    代码省略    }
```

程序员编程时可灵活运用模板技术。在定义多个重载函数时应考虑是否可以将它们合并成一个函数模板，这样可以凝练代码。在定义单个函数时应考虑是否可以将该函数升级成一个函数模板，这样可以提高函数代码的可重用性。

在调用函数模板的程序员看来，函数模板与普通函数没有什么区别。唯一不同的是，函数模板就像是一个具有通用类型的函数，可以处理不同类型的数据。

### 本节习题

1. 下列关于函数模板的描述中，错误的是（　　）。
   A. 函数模板将一组算法相同但所处理数据类型不同的重载函数凝练成一个函数模板
   B. 编译时，由编译器按照函数模板自动生成针对不同数据类型的重载函数定义代码
   C. 定义函数模板以关键字 template 开头
   D. 函数模板不能提高程序代码的可重用性
2. 下列类型参数列表声明中，错误的是（　　）。
   A. <typename T>　　B. <typename TT>　　C. <class T>　　D. <class T, TT>
3. 已定义如下函数模板：

```
template  <typename T>  T Max( T x, T y )  {  …  }
```

   则编译语句“cout << Max(3.5f, 6.2f);”将自动生成下列哪个函数定义代码？（　　）
   A. short Max(short x, short y)　{　…　}
   B. int Max(int x, int y)　{　…　}
   C. float Max(float x, float y)　{　…　}
   D. double Max(double x, double y)　{　…　}
4. 应用函数模板不能实现下列哪种功能？（　　）
   A. 减少源程序代码量　　B. 减少可执行程序代码量
   C. 提高程序代码的可重用性　　D. 函数模板可以像普通函数一样调用

## 10.2　类模板

应用模板技术，也可以将一组功能相同但所处理数据类型不同的类凝练成一个类模板。编译时，再由编译器按照类模板自动生成针对不同数据类型的类定义代码。

### 10.2.1　类模板的定义与使用

**C++语法：定义类模板**

```
template  <类型参数列表>
class  类名  // 类声明部分
{
    类成员声明
}
// 类实现部分：所有类外定义的函数成员，必须按如下的语法形式将它们定义成函数模板
template  <类型参数列表>
函数类型  类名<类型参数名列表> :: 函数名( 形式参数列表 )
{  函数体  }
```

语法说明：

- 定义类模板以关键字 **template** 开头。
- **类型参数列表**可声明一个或多个类型参数，每个类型参数以“typename **类型参数名**”或“class **类型参数名**”的形式声明，类型参数之间用“,”隔开。类型参数名需符合标识符的命名规则。
- 类模板定义的其余部分，包括类名、类成员声明以及类实现部分，它们和普通类的定义形式基本相同。所不同的是，类型参数就像是一种新的数据类型，可以用它来定义数据成员，也可以用来定义函数成员。
- 定义类模板的函数成员，如果在类内（即声明部分）定义，其语法形式与普通类的函数成员没有区别；如果在类外（即类实现部分）定义，则必须按照函数模板的语法形式来定义，并且要在函数名前加“类名<**类型参数名列表**> ::”限定。类型参数名列表应列出类型参数列表中的所有类型参数名，多个参数名之间用“,”隔开。需要注意的是，必须将类模板实现部分的代码与声明部分放在同一个头文件中，不能分开存放。
- 类模板中声明的类型参数是表示数据类型的参数。使用类模板时，类型参数可被替换成任意一种实际数据类型，例如某种基本数据类型、自定义数据类型或类类型等。类型参数也可理解成是一种通用数据类型。

定义好的类模板可像普通类一样被用来定义对象。使用类模板定义对象时，需要明确给出类模板中类型参数所指代的实际数据类型。其语法形式如下：

**类模板名** <**实际数据类型列表**> **对象名 1**, **对象名 2** …… ;

例 10-3 给出两个完整的演示类模板的 C++程序。其中，程序(a)在类内（即声明部分）定义函数成员（内联函数），而程序(b)则演示了如何在类外（即类实现部分）将函数成员定义成函数模板。

**例 10-3 两个演示类模板语法的 C++程序**

**(a) 在类内定义函数成员（内联）**

```
#include <iostream>
using namespace std;

template  <typename T>  // 类模板 A
class A                 // 类声明部分
{
private:   // 声明以下 2 个数据成员
    T  a1;
    int  a2;
public:   // 定义以下 3 个函数成员
    A(T p1, int p2)       // 构造函数
    {  a1 = p1;   a2 = p2;  }
    void Show( )          // 显示数据成员
    {  cout << a1 << ", " << a2 << endl;  }
    T  Sum( )             // 求数据成员的和
    {  return (T)( a1 + a2 );  }
};
```

**(b) 在类外定义函数模板成员**

```
#include <iostream>
using namespace std;

template  <typename T>  // 类模板 A
class  A                // 类声明部分
{
private:   // 声明以下 2 个数据成员
    T  a1;
    int  a2;
public:   // 声明以下 3 个函数成员
    A(T p1, int p2);      // 构造函数
    void Show( );         // 显示数据成员
    T  Sum( );            // 求数据成员的和
};

// 类实现部分：需按函数模板的语法形式定义
template  <typename T>
```

```
18  // 无类实现部分
19
20  int main( )
21  {
22       // 用类模板 A 定义对象，并访问其成员
23       A <double>   o1(10.5, 6);     // double 型对象 o1
24       o1.Show( );                    // 显示：10.5, 6
25       cout << o1.Sum( ) << endl;   // 显示：16.5
26
27       A <int>   o2(10, 6);           // int 型对象 o2
28       o2.Show( );  // 显示：10, 6
29       cout << o2.Sum( ) << endl;   // 显示：16
30       return 0;
31  }
```

```
18  A <T> :: A(T p1, int p2)      // 构造函数
19  {   a1 = p1;   a2 = p2;   }
20
21  template   <typename T>
22  void A <T> :: Show( )       // 显示数据成员
23  {   cout << a1 << " " << a2 << endl;   }
24
25  template   <typename T>
26  T   A <T> :: Sum( )          // 求数据成员的和
27  {   return (T)( a1 + a2 );   }
28
29  int main( )
30  {   // 主函数代码同(a)，省略   }
```

例 10-3 程序的代码说明如下：

- **类模板的定义**。类模板 A 中定义了一个类型参数 T（代码第 4 行）。T 是一种数据类型，可以用它来定义数据成员（例如 a1），也可以用来定义函数成员（例如构造函数和 Sum）。在编写类模板的程序员看来，使用 typename 或 class 所声明的类型参数就像是一种新的数据类型，可以用来定义数据成员，也可以用来定义函数成员，例如用它来定义函数类型（即返回值的类型），也可以用来定义形参或局部变量。
- **类模板的使用**。定义好的类模板可像普通类一样被用来定义对象。使用类模板定义对象时，需要明确给出类模板中类型参数 T 所指代的实际数据类型。例如，给出 double 型就可以定义 double 型 A 类对象 o1（程序(a)第 23 行），或给出 int 型就可以定义 int 型 A 类对象 o2（程序(a)第 27 行）。在调用者看来，类模板中的类型参数就像是一种通用数据类型。

类模板为什么能像普通类一样定义对象呢，其中类型参数的作用又是什么呢？回答这两个问题首先需要了解类模板的编译原理。

## 10.2.2 类模板的编译原理

例 10-3 的类模板 A 定义了一个类型参数 T。T 是表示数据类型的参数，可指代任意一种实际数据类型。使用 C++编译器编译例 10-3(a)的 C++源程序，当编译到使用类模板定义对象语句时，编译器将根据所给出的实际数据类型来取代类型参数 T。例如：

- 编译例 10-3(a)中代码第 23 行的定义对象语句：

```
A <double>   o1(10.5, 6);
```

该语句使用类模板 A 定义对象 o1，定义时给出了实际数据类型 double。编译该语句，编译器首先用数据类型 double 取代类模板 A 中所有的类型参数 T，从而自动生成一个 double 型类 A。double 型类 A 是一个普通的类。最终编译器是用这个 double 型类 A 来定义对象 o1。

编译时将类模板中类型参数绑定到某个具体数据类型的过程，称为类模板的**实例**

**化**。实例化所生成的类称为类模板的**实例类**。实例类是一个普通的类，可以用来定义对象。

■　编译例 10-3(a)中代码第 27 行的定义对象语句：

```
A <int>   o2(10, 6);
```

该语句使用类模板 A 定义对象 o2，定义时给出了实际数据类型 int。编译该语句，编译器将再次使用类模板 A 进行实例化，自动生成一个 int 型类 A。然后再用这个 int 型类 A 定义出对象 o2。

类模板的**编译原理**是：类模板是具有类型参数的类。类型参数是表示数据类型的参数，可指代任意一种实际数据类型。编译器在编译到使用类模板定义对象语句时，将首先按照所给定的实际数据类型对类模板进行实例化，生成一个实例类。最终，编译器使用实例类来定义所需要的对象。

使用类模板定义对象，编译器会**隐含地**实例化类模板。也可以使用 typedef 类型定义来**显式地**实例化类模板。例如，可以修改例 10-3(a)中的主函数，使用 typedef 类型定义来实例化类模板 A。

```
int main( )
{
    typedef   A<double>   DoubleA;       // 实例化类模板 A，生成实例类 DoubleA
    DoubleA   o1(10.5, 6);               // 定义类 DoubleA 的对象 o1
    o1.Show( );                          // 显示：10.5, 6
    cout << o1.Sum( ) << endl;           // 显示：16.5

    typedef   A<int>   IntA;             // 实例化类模板 A，生成实例类 IntA
    IntA   o2(10, 6);                    // 定义类 IntA 的对象 o2
    o2.Show( );                          // 显示：10, 6
    cout << o2.Sum( ) << endl;           // 显示：16
    return 0;
}
```

将一组功能相同但所处理数据类型不同的类凝练成一个类模板，这样可以精简源代码，减少程序员的编码工作量。但使用类模板不会减少所编译出的可执行代码量，程序的执行效率也保持不变。

### 10.2.3　类模板的继承与派生

类模板可以被继承，派生出新类。以类模板为基类定义派生类，可以在派生时实例化，也可以继续定义派生类模板。

#### 1. 定义实例化派生类

例 10-4 给出一个实例化派生类的 C++演示程序，其中的基类 Base 是一个类模板，派生类 Derived 继承该类模板。派生类 Derived 对基类 Base 进行实例化，用 double 类型取代类型参数 T。派生类 Derived 是一个实例类，可以像普通类一样来定义对象。

**例 10-4 一个实例化派生类的 C++演示程序**

**(a) 基类 Base：类模板**

```
#include <iostream>
using namespace std;

// 基类 Base：是一个类模板
template  <typename T>
class  Base  // 声明部分
{
private:
    T  a;  // 数据成员
public:
    Base(T  x)  // 构造函数
    {  a = x;  }
    void Show( )  // 显示数据成员
    {
        Cout << "a=" << a << ", ";
    }
};
```

**(b) 派生类 Derived：实例化类**

```
// 派生类 Derived：公有继承类模板 Base，派生时实例化
class  Derived : public Base <double>
{
private:
    int  b;  // 新增数据成员
public:
    // 注意派生类构造函数的写法
    Derived(double p1, int p2) : Base <double>( p1 )
    {  b = p2;  }
    void Show( )  // 新增函数成员 Show
    {
        Base <double>::Show( );  // 调用基类同名函数
        cout << "b=" << b << endl;
    }
};

int main( )
{
    Derived  obj(10.5, 6);  // 定义派生类 Derived 的对象 obj
    obj.Show( );  // 显示结果：a=10.5, b=6
    return 0;
}
```

在编译到派生类 Derived 的定义代码时，编译器将按照所给定的实际数据类型 double 对类模板 Base 进行实例化，生成一个 double 型的实例类，最终派生类 Derived 继承的是该实例类。

### 2. 定义派生类模板

例 10-5 给出一个派生类模板的 C++演示程序，其中的基类 Base 和例 10-4 一样，是类模板。所不同的是，派生类 Derived 仍然是一个类模板。

**例 10-5 一个派生类模板的 C++演示程序**

**(a) 基类 Base：类模板**

```
#include <iostream>
using namespace std;

// 基类 Base：是一个类模板
template  <typename T>
class  Base  // 声明部分
{
private:
    T  a;  // 数据成员
public:
    Base(T  x)  // 构造函数
```

**(b) 派生类 Derived：类模板**

```
// 派生类 Derived：公有继承类模板 Base，派生类仍为类模板
template  <typename T, typename TT> // 新增类型参数 TT
class  Derived : public Base <T>
{
private:
    TT  b;  // 新增数据成员
public:
    // 注意派生类构造函数的写法
    Derived(T  p1, TT  p2) : Base <T>( p1 )
    {  b = p2;  }
    void Show( )  // 新增函数成员 Show
```

```
12        {   a = x;   }
13        void Show( )    //  显示数据成员
14        {
15            cout << "a=" << a << ", ";
16        }
17    };
```

```
12        {
13            Base <T>::Show( );    //  调用基类同名函数
14            cout << "b=" << b << endl;
15        }
16    };
17
18    int main( )
19    {
20        //  使用派生类模板 Derived 定义对象 obj
21        Derived<double, int>    obj(10.5, 6);
22        obj.Show( );    //  显示结果：a=10.5, b=6
23        return 0;
24    }
```

派生类 Derived 在继承基类 Base 时，又新增了一个类型参数 TT（程序(b)代码第 2 行）。和其他类模板一样，可以使用派生类模板 Derived 定义对象。定义时需要明确给出派生类模板中类型参数所指代的实际数据类型（程序(b)代码第 21 行）。

程序员编程时可灵活运用模板技术。在定义多个功能相同但所处理数据类型不同的类时应考虑是否可以将它们合并成一个类模板，这样可以凝练代码。在定义单个类时应考虑是否可以将它升级成一个类模板，这样可以提高类代码的可重用性。

在使用类模板的程序员看来，类模板与普通类没有什么太大区别。只是在使用类模板时需要给出类型参数所指代的实际数据类型。

## 本节习题

1. 下列关于类模板的描述中，错误的是（　　）。
   A. 类模板将一组功能相同但所处理数据类型不同的类凝练成一个类模板
   B. 编译时由编译器按照类模板自动生成针对不同数据类型的实例类
   C. 定义类模板以关键字 class 开头
   D. 类模板可以提高程序代码的可重用性
2. 已定义如下的类模板：

```
template    <typename T, typename TT>
class ABC
{   …   };
```

   则下列对象定义语句中，错误的是（　　）。
   A. ABC <int, char>　obj;　　B. ABC <short, long>　obj;
   C. ABC <double, double>　obj;　　D. ABC <double>　obj;
3. 已定义如下类模板：

```
template    <typename T>    class ABC    {    …    };
```

   则下列哪条语句是错误的？（　　）
   A. ABC　obj;
   B. typedef　ABC <double>　DoubleABC;

C. class Derived : public ABC <double> { … };
D. template <typename T, typename TT> class Derived : public ABC <T> { … };

4. 下列关于类模板的描述中，错误的是（ ）。
A. 类模板可以减少源程序代码量　　B. 类模板可以提高程序代码的可重用性
C. 类模板可以被实例化　　D. 类模板不能被继承

## 10.3 C++标准库

为了方便程序员，C++语言以标准库的形式为程序员提供了很多常用的函数和类。按功能来划分，C++标准库共提供了 10 大类功能。

- **语言支持**（language support）类。为 C++语言的动态内存分配（例如 new/delete 运算符）、异常处理机制（try-catch）等提供底层支持。
- **通用工具**（general utilities）类。为标准库中的其他功能类提供底层支持。
- **输入/输出**（input/output）类。通过输入/输出流类族提供标准 I/O、文件 I/O 和字符串 I/O 的功能。
- **字符串**（string）类。通过字符串类 string、宽字符串类 wstring 等提供常用的文字处理功能。
- **诊断**（diagnostics）类。通过异常类族（基类为 exception），为 C++语言异常处理机制提供多种不同功能的异常类。
- **容器**（container）类、**迭代器**（iterator）和**算法**（algorithm），它们为数据集合提供了常用的存储类和处理算法。
- **数值**（numerics）类。为更高级的数值计算（例如并行计算）提供底层支持，另外还提供了一个复数类 complex。
- **本地化**（locale）类。为文字处理提供国际化支持。

C++标准库不是 C++语言本身的内容，是其附属组成部分。C++标准库极大地扩展了 C++语言的功能，让 C++程序员能够站在更高的起点上开始编程。C++程序员应尽可能使用标准库，这样可以提高开发效率，同时也能提高程序的可靠性和执行速度。

为了凝练代码，C++标准库广泛采用了模板技术，因此 C++标准库有时也被称作标准模板库 STL（Standard Template Library）。例如，C++标准库中的流类库就使用了模板技术。以 ios 为基类的流类库只能对基于 ANSI 编码的字符数据进行输入/输出。为了适应 Unicode 编码，C++流类库另外定义了一套基于 Unicode 编码的流类库。这两套流类库功能相同，仅仅是所处理的数据类型不同，其中一个是字符类型 char，另一个是宽字符类型 wchar_t。使用模板技术，C++标准库将这两套类库合并成了一套类模板，其示意代码如下：

```
namespace std // 命名空间 std
{
    // 输入/输出流类模板（此处仅列出类模板的声明）
    template <class charT> class basic_ios;
    template <class charT> class basic_istream;
    template <class charT> class basic_ostream;
```

```
    template <class charT> class    basic_iostream;
    template <class charT> class    basic_ifstream;
    template <class charT> class    basic_ofstream;
    template <class charT> class    basic_fstream;
    template <class charT> class    basic_istringstream;
    template <class charT> class    basic_ostringstream;
    template <class charT> class    basic_stringstream;
    // 使用类模板定义基于 ANSI 编码的输入/输出流类
    typedef   basic_ios<char>    ios;
    typedef   basic_istream<char>    istream;
    typedef   basic_ostream<char>    ostream;
    typedef   basic_iostream<char>    iostream;
    typedef   basic_ifstream<char>    ifstream;
    typedef   basic_ofstream<char>    ofstream;
    typedef   basic_fstream<char>    fstream;
    typedef   basic_istringstream<char>    istringstream;
    typedef   basic_ostringstream<char>    ostringstream;
    typedef   basic_stringstream<char>    stringstream;
    // 使用类模板定义基于 Unicode 编码的输入/输出流类
    typedef   basic_ios<wchar_t>    wios;
    typedef   basic_istream<wchar_t>    wistream;
    typedef   basic_ostream<wchar_t>    wostream;
    typedef   basic_iostream<wchar_t>    wiostream;
    typedef   basic_ifstream<wchar_t>    wifstream;
    typedef   basic_ofstream<wchar_t>    wofstream;
    typedef   basic_fstream<wchar_t>    wfstream;
    typedef   basic_istringstream<wchar_t>    wistringstream;
    typedef   basic_ostringstream<wchar_t>    wostringstream;
    typedef   basic_stringstream<wchar_t>    wstringstream;
}
```

因为使用了模板技术，C++标准库能以较少的代码量提供很强大的功能。本书第 9 章已学习了输入/输出流类和字符串类。本章下面的内容是结合标准库，重点介绍 C++语言的异常处理机制和数据集合的存储及处理方法。

## 本节习题

1. 下列关于 C++标准库的描述中，错误的是（　　）。
   A. C++语言以标准库的形式为程序员提供了很多常用的函数和类
   B. C++标准库扩展了 C++语言的功能，使程序员可以在更高的起点上开发程序
   C. C++标准库在编写时没有采用模板技术
   D. 流类库是 C++标准库的组成部分
2. C++标准库没有包含下列哪种类？（　　）
   A. 字符串（string）类　　B. 诊断（diagnostics）类
   C. 容器（container）类　　D. 圆形（circle）类

# 10.4 C++语言的异常处理机制

编写程序过程中出现错误是难免的。程序错误可分为三类，它们分别是**语法错误**、**语义错误**（或称逻辑错误）和**运行时错误**。针对不同错误，C++语言具有不同的解决办法，最终保证所开发的程序能够正确、稳定地运行。

针对程序运行时错误 C++语言设计了专门的异常处理机制，即 try-catch 机制。C++标准库为异常处理机制提供了多种不同功能的异常类。

## 10.4.1 程序中的三类错误

本节通过具体的程序实例介绍语法错误、语义错误和运行时错误及其对应的解决办法。

### 1. 语法错误

假设编写一个简单的除法运算程序，计算 100 个苹果被 N 个人分，每个人能分几个。例 10-6 给出一个简单的 C++除法运算程序。其中，代码第 10 行有一个语法错误。键盘对象 cin 输入数据不能用插入运算符<<，而应使用提取运算符>>。

**例 10-6 一个简单的 C++除法运算程序（存在语法错误）**

```
1   #include <iostream>
2   using namespace std;
3
4   int   Div( int n )                          // 返回 100 ÷ n 的结果
5   {   return   ( 100 / n ) ;   }
6
7   int   main( )
8   {
9       int   N;                                // 定义变量 N，保存键盘输入的人数
10      cin << N;                               // 语法错误：输入 N 的值，正确语法应为：cin >> N;
11      int   result = Div( N );                // 调用函数 Div 进行除法运算
12      cout << "100÷" << N << "=" << result << endl;       // 显示 100 ÷ N 的结果
13      return 0;
14  }
```

使用 VC 6.0 编译例 10-6 的程序，编译器将提示如下错误信息：

'<<': class istream does not define this operator

其含义是：istream 类未定义插入运算符<<。程序员应按照错误提示，将插入运算符<<改为提取运算符>>，这样就完成了语法错误的修改。再次编译修改后的程序，没有任何语法错误，编译通过。

如 C++程序员未能严格按照语法规则编写程序，这就属于语法错误。编译时，C++编译器负责检查源程序中的语法错误，如无语法错误则将其翻译成目标程序，否则将提示错误信息。C++编译器能够帮助程序员检查出所有的语法错误，因此语法错误易于检查，易

于修改。

### 2. 语义错误

同样的除法运算，例 10-7 给出另一份 C++程序代码。其中，代码第 5 行有一个语义错误。本应是除法运算却被错误地写成了乘法运算，语法正确但语义错误。

**例 10-7　一个简单的 C++除法运算程序（存在语义错误）**

```
1  #include <iostream>
2  using namespace std;
3
4  int   Div( int n )                      // 返回 100 ÷ n 的结果
5  {   return   ( 100 * n ) ;   }          // 语义错误：将除法错误地写成了乘法，语法正确但语义错误
6
7  int   main( )
8  {
9       int   N;                           // 定义变量 N，保存键盘输入的人数
10      cin >> N;
11      int   result = Div( N );           // 调用函数 Div 进行除法运算
12      cout << "100÷" << N << "=" << result << endl;          // 显示 100 ÷ N 的结果
13      return 0;
14 }
```

编译例 10-7 的程序，无任何语法错误，编译通过。但运行该程序，输入人数 2：

**2**<回车键>

程序将显示如下结果：

100÷2=200

上述结果显然是错误的，正确结果应为：

100÷2=50

运行测试的结果表明，程序中存在语义错误。程序员需检查源程序，找出错误原因。本例中，将代码第 5 行的“100 * n”改成“100 / n”，这样就完成了对语义错误的修改。

C++编译器不能帮助程序员发现语义错误。程序员必须通过运行测试，比对程序结果才能发现语义错误。

### 3. 运行时错误

同样的除法运算，例 10-8 给出一份既没有语法错误，也没有语义错误的 C++程序代码。

**例 10-8　一个简单的 C++除法运算程序（无任何语法或语义错误）**

```
#include <iostream>
using namespace std;

int   Div( int n )                      // 返回 100 ÷ n 的结果
```

```
{   return   ( 100 / n ) ;   }

int   main( )
{
    int   N;                          // 定义变量 N，保存键盘输入的人数
    cin >> N;
    int   result = Div( N );          // 调用函数 Div 进行除法运算
    cout << "100÷" << N << "=" << result << endl;       // 显示 100 ÷ N 的结果
    return 0;
}
```

编译例 10-8 的程序，无任何语法错误，编译通过。假设该程序所生成的可执行程序为 test.exe，在 Windows 操作系统上运行该程序，输入人数 2：

2<回车键>

程序将显示如下结果：

100÷2=50

运行结果也正确。但再次运行该程序，输入人数 0：

0<回车键>

这时，Windows 操作系统将提示该程序运行出现严重错误，已被停止运行（如图 10-1 所示）。因为任何数都不能被零除，计算机无法执行“100/0”这样的运算，因此将停止整个程序的运行。类似这种因用户输入不当而导致的程序运行错误就是一种运行时错误。

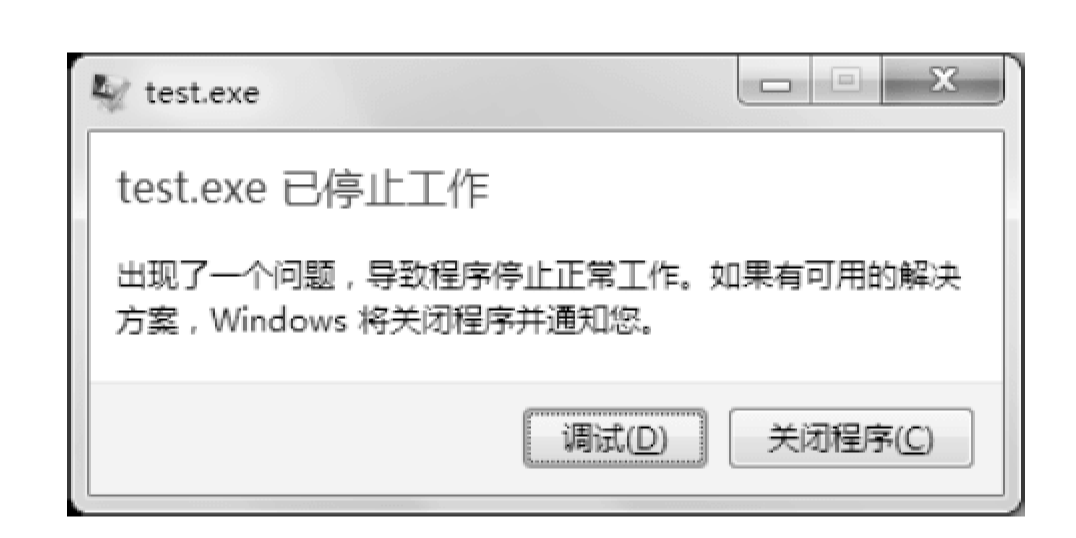



图 10-1　例 10-8 可能导致的程序运行时错误

程序运行时，因运行环境差异或用户操作不当所造成的程序错误被统称为**运行时错误**。运行环境存在差异，或用户出现操作不当，这些都被称为程序运行时的**异常情况**。程序员无法阻止异常情况的出现，但可以在程序中添加**异常处理机制**，避免因异常情况而导致程序死机等严重的运行时错误。

## 10.4.2　程序异常处理机制

程序运行时，常见的异常情况有：

■ **用户操作不当**。例如，运行例 10-8 除法运算程序时输入了人数 0。

■ **输入文件不存在**。从文件中输入数据，但文件不存在，打开（open）文件将会失败。如果文件未能成功打开却继续对文件进行读写操作，这会导致运行时错误。

■ **网络连接中断**。程序可以通过网络发送/接收数据。在网络连接中断的情况下，继续发送/接收数据将导致运行时错误。

■ **非法访问内存单元**。因异常情况导致指针变量未能正确赋值，此时通过指针变量访问内存单元可能会导致运行时错误。

程序员无法阻止上述异常情况的发生，但能够预见到程序运行时可能发生哪些异常。程序员应当在程序中添加异常处理机制，以避免程序运行时因异常情况而导致的死机或意外中止等严重错误。

一个**异常处理机制**由如下两部分组成。

（1）**发现异常**。程序员应在可能出现异常的程序位置增加检查异常的代码，其目的就是及时发现异常。

（2）**处理异常**。发现异常后，程序需要改变算法流程，否则将产生运行时错误。处理异常就是在算法中增加异常处理流程。没有异常时，程序执行正常算法流程；发现异常时，程序执行异常处理流程。

为例 10-8 的 C++除法运算程序添加异常处理机制，具体代码如例 10-9 所示。其中加粗部分是新添加的异常处理代码。

**例 10-9　一个简单的 C++除法运算程序（具有异常处理机制）**

```
#include <iostream>
using namespace std;

int   Div( int n )                     // 返回 100 ÷ n 的结果
{
      if ( n <= 0 )                    // 检查异常：如果 n<=0，则属于异常情况
            return   ( –1 );           // 执行异常处理流程。本例直接返回–1
      // 如果没有异常，则执行如下的正常处理流程
      return   ( 100 / n ) ;
}

int   main( )
{
      int   N;                         // 定义变量 N，保存键盘输入的人数
      cin >> N;
      int   result = Div( N );         // 调用函数 Div 进行除法运算
      if ( result == -1 )              // 检查函数 Div 的返回值：–1 表示函数 Div 出现了异常
            cout << "输入的人数必须为正整数" << endl;   // 向用户显示错误信息
      else
            cout << "100÷" << N << "=" << result << endl;   // 显示 100 ÷ N 的结果
      return 0;
}
```

例 10-9 程序的代码说明如下。

**1）被调函数 Div**

被调函数 Div 的功能是计算 100 个苹果被 n 个人分，每个人能分几个。合理的形参 n 应为正整数，否则属于异常情况，即用户输入了错误的数据。增加一条 if 语句，用于检查异常（代码第 6 行）。如果 n 为正整数，则表明没有异常，程序执行正常算法流程。如果 n 为负数或 0，则表明发现了异常。发现异常后该如何处理呢？函数 Div 的做法是通过特定的返回值−1 向上级主调函数 main 报告异常（代码第 7 行）。

通过特定的返回值向上级主调函数报告异常，再由上级主调函数具体处理异常，这是一种常规的异常处理方法，并在结构化程序设计方法和 C 语言中广泛使用。表示函数出现异常的特定返回值被称为函数的**错误代码**。程序员可以定义不同的错误代码来表示不同的异常。

**2）主调函数 main**

主调函数 main 接收被调函数 Div 的返回值，并再次增加一条 if 语句，用于检查调用函数 Div 时是否出现了异常（代码第 17 行）。如果返回值为−1，则表明发现了异常，主函数使用 cout 语句显示错误信息，提示用户错误原因（代码第 18 行）。

运行该程序，输入人数 2：

```
2<回车键>
```

程序将显示如下结果：

```
100÷2=50
```

运行结果也正确。但再次运行该程序，输入人数 0：

```
0<回车键>
```

程序将显示如下结果：

```
输入的人数必须为正整数
```

通过例 10-9 可以看到，在多函数结构程序中运用异常处理机制，将会同时涉及主调函数与被调函数。被调函数负责发现异常，发现异常后应处理异常，并向上级主调函数报告异常。常规的报告方法是通过特定的函数返回值（即错误代码）来表示不同的异常情况。主调函数通过被调函数的返回值来检查调用时是否出现了异常，发现异常后也需要做相应的异常处理。如果是函数多级调用，主调函数还可能需要逐级向各自的上级函数报告异常。

涉及主调函数与被调函数之间的异常处理机制被称为**多级异常处理机制**。多级异常处理机制包括三个方面的内容，它们分别是发现异常、报告异常和处理异常。C++语言为多级异常处理提供了一种非常方便而有效的机制，这就是 try-catch 异常处理机制。

### 10.4.3 try-catch 异常处理机制

C++语言通过 **throw** 语句报告异常，并通过 **try-catch** 语句提供一个捕获并处理异常的代码框架。C++程序使用上述两条语句就可以实现一套完整的异常处理机制。

1. 语法规则

C++语法：throw 语句

```
throw 异常表达式 ;
```

语法说明：

- **异常表达式**的结果可以是基本数据类型、自定义数据类型或类类型。异常表达式结果的数据类型被用于区分不同错误引起的异常，结果的值被用于描述异常的详细信息。异常表达式可以是单个常量、变量或对象。
- 计算机执行该语句，将抛出一个异常，并退出当前函数的执行。throw 语句的功能是报告异常，该异常将被 try-catch 语句捕获并做相应的异常处理。

举例：

```
throw   15;                // 抛出一个 int 型异常，该异常的详细信息为 15
throw   "Invalid value";   // 抛出一个字符串型异常，该异常的详细信息为“Invalid value”
```

C++语法：try-catch 语句

```
try
{
    受保护代码段
}
catch ( 异常类型 1 )
{  异常类型 1 的处理代码 }
catch ( 异常类型 2 )
{  异常类型 2 的处理代码 }
……
```

语法说明：

- 如果预计某个程序代码段有可能发生异常，程序员可使用 try **子句**将该代码段保护起来。受保护代码段在执行时将启用 C++语言的异常保护机制，捕获该代码段执行过程中的任何异常报告，包括下级被调函数所报告的异常。
- catch **子句**负责捕获并处理异常。每个 catch 子句只负责处理一种类型的异常。异常类型就是抛出该异常的 throw 语句中表达式结果的类型。
- 若受保护代码段在执行过程中未报告任何异常，则所有的 catch 子句都不会执行。
- 若受保护代码段在执行过程中有异常，则根据异常类型依次匹配 catch 子句。如果匹配上某个 catch 子句（称异常被捕获），则执行所匹配 catch 子句的处理代码。每个异常最多只会有一个 catch 子句的处理代码被执行，其他 catch 子句都不会执行。如果异常未被任何 catch 子句捕获，则自动逐级向上级函数继续报告该异常，直到被上级函数中的某个 catch 子句所捕获。假设某个异常，连最上级的主函数 main 也未能捕获它，也就是说没有任何函数能够捕获并处理该异常，则自动使用**默认方法**来进行处理。默认的异常处理方法通常是中止当前程序的执行，并提示简单的错误信息。
- **catch( ... )**形式的子句（注：括号中的“...”是由三个点组成的）可匹配并捕获任意类型的异常，其后面的 catch 子句都将是无效子句。因此 catch( ... )形式子句应放在所有 catch 子句的最后。

使用 C++语言的 try-catch 异常处理机制，对例 10-8 的 C++除法运算程序进行保护，具体代码如例 10-10 所示。其中加粗字体部分是新添加的异常处理代码。

**例 10-10　一个简单的 C++除法运算程序（具有 try-catch 异常处理机制）**

```
1   #include <iostream>
2   using namespace std;
3
4   int   Div( int n )                      // 返回 100 ÷ n 的结果
5   {
6       if ( n <= 0 )                       // 检查异常：如果 n<=0，则属于异常情况
7           throw   (–1 );   // 执行异常处理流程：抛出 int 型异常–1，并退出当前函数的执行
8       // 如果没有异常，则执行如下的正常处理流程
9       return   ( 100 / n ) ;
10  }
11
12  int   main( )
13  {
14      int   N;                            // 定义变量 N，保存键盘输入的人数
15      cin >> N;
16      try
17      {
18          int   result = Div( N );        // 调用函数 Div 进行除法运算
19          cout << "100÷" << N << "=" << result << endl;   // 显示 100 ÷ N 的结果
20      }
21      catch ( int )   // 捕获 int 型异常
22      {
23          cout << "输入的人数必须为正整数" << endl;   // 向用户显示错误信息
24      }
25      catch ( ... )   // 捕获任意类型的异常
26      {
27          cout << "发生了其他异常" << endl;           // 向用户显示错误信息
28      }
29      return 0;
30  }
```

例 10-10 程序的代码说明如下。

**1）try 子句**

例 10-10 中代码第 18~19 行是受 try 子句保护的代码段。该代码段在执行过程中发生的任何异常都将交由其后的 catch 子句进行处理，包括调用函数 Div(N)过程中所发生的异常。

**2）throw 语句**

函数 Div(N)如发现 n<=0，就使用 throw 语句抛出一个异常（代码第 7 行），其中−1 是一个 int 型异常。类似于 return 语句，计算机执行 throw 语句时将首先计算异常表达式，然后抛出异常并退出当前函数的执行。throw 语句与 return 语句不同的是，执行 return 语句将返回主调函数，继续执行函数调用语句的下一条语句，而执行 throw 语句将直接跳转到 try 子句后面的 catch 子句，开始执行 catch 子句的异常捕获及处理操作；return 语句是正常退出函数，而 throw 语句则是异常退出。try-catch 机制保证异常退出时能自动释放当前函数

的形参、局部变量或对象。释放局部对象时会自动调用其析构函数。

**3）catch 子句**

例 10-10 中代码第 21~28 行有两个 catch 子句。第一个 catch 子句专门捕捉 int 类型的异常。第二个 catch 子句使用了“...”，它可以捕捉到所有其他类型的异常。catch 子句在捕捉到异常之后，将执行其异常处理代码。每个异常只有第一个与其类型匹配的 catch 子句会被执行。执行结束后将直接跳转到 catch 子句的最后，继续执行下一条语句（本例中为第 29 行的 return 语句）。

本例中代码第 7 行所抛出的−1 是一个 int 型异常，将被第 21 行的 catch( int )子句捕获，然后执行其异常处理代码（代码第 22~24 行），显示错误信息“输入的人数必须为正整数”。执行完异常处理代码后跳转到第 29 行，执行 return 语句，程序结束。

### 2. 在 catch 子句中接收异常的详细信息

catch 子句可以定义参数，用于接收异常的详细信息。例如，将例 10-10 第 21~24 行的 catch 子句改写成如下的形式：

```
catch ( int x )                                        // 定义参数 x 接收异常的详细信息
{
    cout << "输入的人数必须为正整数" << endl;          // 向用户显示错误信息
    cout << "异常的详细信息：" << x << endl;           // 显示异常的详细信息
}
```

运行修改后的程序，输入人数 0：

```
0<回车键>
```

程序将显示如下结果：

```
输入的人数必须为正整数
异常的详细信息：–1
```

可以看出，catch 子句所定义的参数相当于函数形参，可以接收 throw 语句中异常表达式的结果。throw 语句中的异常表达式相当于是调用函数时的实参。

### 3. 使用类描述异常

throw 语句所抛出的异常可以是基本数据类型、自定义数据类型或类类型。类类型的异常可以提供更多的异常信息和异常处理功能。修改例 10-10，将 throw 语句所抛出的异常由整数−1 改成一个异常类 Error 的对象 err，具体代码如例 10-11 所示。

**例 10-11　一个简单的 C++除法运算程序（使用异常类）**

```
#include <iostream>
using namespace std;

class   Error                              // 异常类 Error
{
public:
```

```
7       int errCode;                          // 异常代码
8       char errMsg[40];                      // 异常信息
9       Error( int code, char *msg )          // 构造函数
10      {   errCode = code;   strcpy( errMsg, msg );   }
11      void ShowError( )                     // 显示异常的详细信息
12      {   cout << errCode << ": " << errMsg << endl;   }
13  };
14
15  int  Div( int n )                         // 返回 100 ÷ n 的结果
16  {
17      if ( n <= 0 )                         // 检查异常：如果 n<=0，则属于异常情况
18      {
19          Error   err( -1, "输入的人数必须为正整数" );   // 定义异常类 Error 的对象 err
20          throw   ( err );                   // 抛出异常对象 err，然后退出当前函数的执行
21      }
22      // 如果没有异常，则执行如下的正常处理流程
23      return   ( 100 / n ) ;
24  }
25
26  int  main( )
27  {
28      int   N;                              // 定义变量 N，保存键盘输入的人数
29      cin >> N;
30      try
31      {
32          int   result = Div( N );          // 调用函数 Div 进行除法运算
33          cout << "100÷" << N << "=" << result << endl;   // 显示 100 ÷ N 的结果
34      }
35      catch ( Error e )                     // 捕捉 Error 类的异常，并定义参数 e 来接收异常对象
36      {
37          e.ShowError( );                   // 向用户显示所捕捉到异常对象 e 的详细信息
38      }
39      return 0;
40  }
```

例 10-11 程序的代码说明如下。

**1）异常类 Error**

例 10-11 首先定义一个异常类 Error（代码第 4~13 行），其中既包含异常代码 errCode，又包含异常信息 errMsg，并提供了构造函数和显示异常信息的函数 ShowError。

**2）抛出异常类对象**

函数 Div 在检测到异常之后，首先构造一个异常类 Error 的对象 err（代码第 19 行），将其异常代码初始化为–1，异常信息初始化为“输入的人数必须为正整数”，然后抛出该异常对象 err（代码第 20 行）。异常类对象可以报告更多的异常信息。

**3）捕捉异常**

本例第 35 行的 catch( Error e )子句可以捕捉所有 Error 类的异常。通过定义参数 e，还可以接收具体的异常对象。通过该异常对象可以获得有关异常的进一步信息，例如代码第 37 行调用异常对象 e 的函数成员 ShowError，显示出如下的错误信息：

```
–1：输入的人数必须为正整数
```

catch 子句所定义的参数也可以是异常类的引用，例如：

```
catch ( Error &re )          // 捕捉 Error 类异常，定义引用参数 re 来访问所捕捉到的异常对象
{
    re.ShowError( );         // 通过引用参数 re 来间接访问所捕捉到异常对象的详细信息
}
```

#### 4. try-catch 异常处理机制的代码框架

图 10-2 给出一个 try-catch 异常处理机制的代码框架示意图。下面结合这个代码框架对 try-catch 异常处理机制做进一步说明。

try-catch 异常处理机制的语法细节：

（1）语句 try-catch 可以多层嵌套。

（2）计算机执行 throw 语句，将跳转到包含该 throw 语句最下一层的 catch 子句进行异常类型匹配，例如图 10-2 中的 throw ①。如果类型匹配成功，则捕获异常并转入异常处理；如果匹配不成功，则逐级跳转到上层的 catch 子句继续类型匹配；如果本程序未能捕获异常，则交由操作系统处理。操作系统通常是中止当前程序的执行，并提示简单的错误信息。

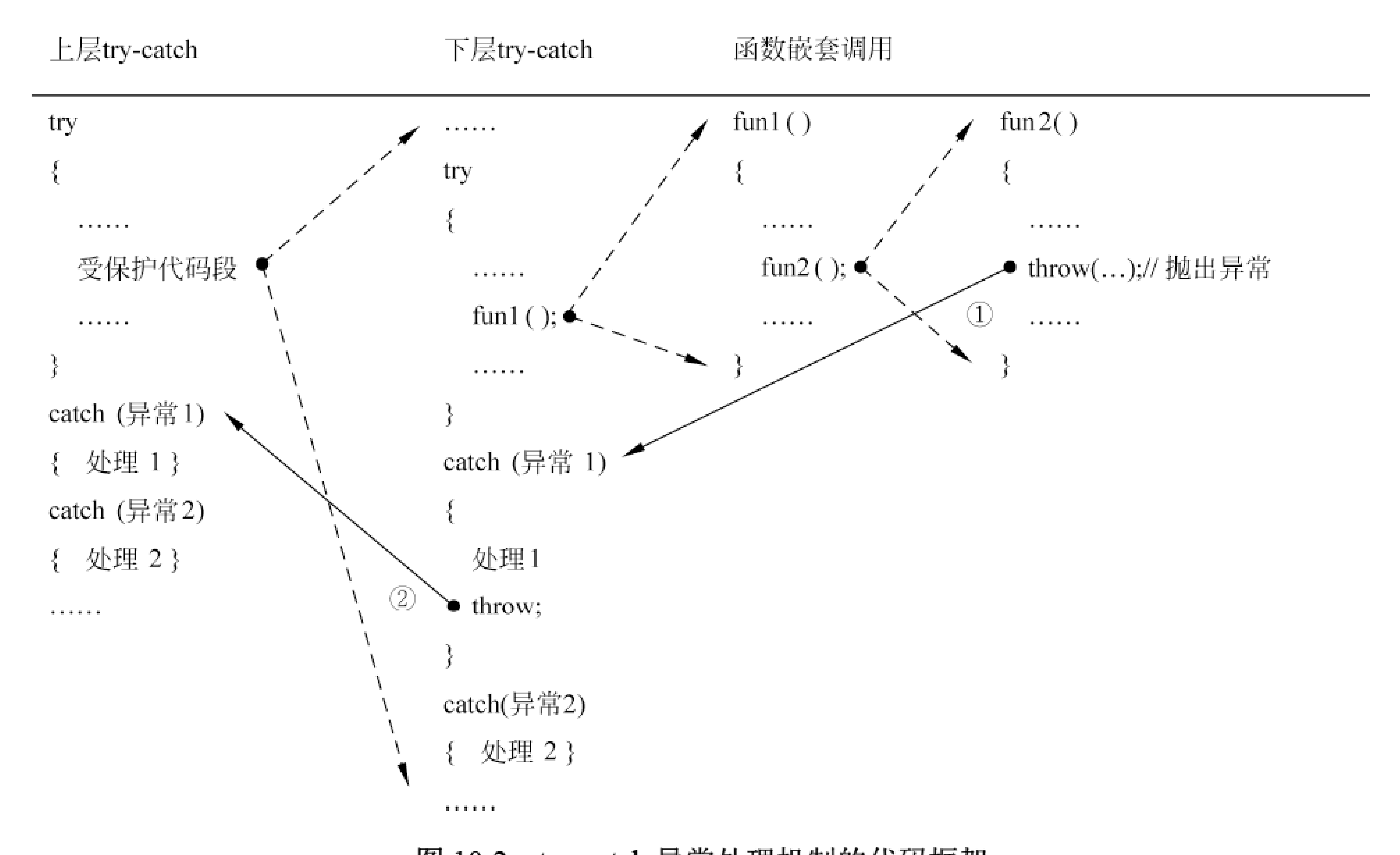



图 10-2 try-catch 异常处理机制的代码框架

（3）语句 try-catch 可以直接包含 throw 语句。例如：

```
try
{
    受保护代码段
    throw –1;                // 执行该语句将跳转到 catch 子句，这时不会退出当前函数
    ……
}
```

```
catch ( int )
{  处理 int 型异常  }
……
```

可以看出，执行 throw 语句时的跳转目标实际上是查找最近的 catch 子句。

（4）子句 catch 在处理完异常后可以继续抛出异常。例如图 10-2 下层 try-catch 中的第一个 catch 子句：

```
catch( 异常 1 )          // 捕捉异常 1
{
    ……                   // 处理异常 1
    throw ;              // 再次抛出当前捕捉到的异常 1，参见图 10-2 中的 throw ②
}
```

再次抛出异常的原因是，某些异常需要经过不同层级代码的多次处理，每个层级只完成整个异常处理环节的一部分。将当前捕捉到的异常再次抛给上层的 try-catch，此时 throw 语句可以不带异常表达式。

关键字 throw 还可以被用在函数的定义或原型声明中，其作用是声明该函数可能抛出的异常列表。其语法形式如下：

**函数类型　函数名(形式参数列表) throw (异常类型列表)**

例如下列声明函数 fun 原型的语句：

```
void fun( ) throw ( A, B, C ) ;    // 函数 fun 能且只能抛出 A、B 或 C 共三种类型的异常
void fun( ) throw ( ) ;            // 函数 fun 不会抛出任何类型的异常
void fun( ) ;                      // 函数 fun 可能会抛出任何类型的异常
```

## 10.4.4　C++标准库中的异常类 exception

为方便程序员，C++标准库预定义了以标准异常类 exception 为基类的异常类族。程序员可直接使用这些异常类，或基于这些异常类派生自己的异常类。另外，C++标准库其他组件所抛出的异常也都来自于这个异常类族。使用 C++标准库中的异常类族须包含相应的类声明头文件。例如，使用标准异常类 exception 要包含如下的类声明头文件：

```
#include <exception>
```

### 1. 标准异常类 exception

例 10-12 给出了标准异常类 exception 的示意代码。

**例 10-12　标准异常类 exception 的示意代码**

```
class exception
{
private:    char *msg ;               // 保存异常信息的数据成员
public:
    exception( );                     // 构造函数
    exception( const char * );
```

```
    exception( const exception & );
    exception & operator=(const exception & );          // 重载赋值运算符
    virtual ~exception( );                               // 析构函数
    virtual const char *what( );                         // 函数成员：返回异常信息
};
```

### 2. 使用标准异常类 exception

可直接使用标准异常类 exception 定义对象。修改例 10-11 的除法运算程序，用标准异常类 exception 代替自定义的异常类 Error。具体代码如例 10-13 所示。

**例 10-13　一个简单的 C++除法运算程序（使用标准异常类 exception）**

```
#include <iostream>
#include <exception>
using namespace std;

int  Div( int n )                    // 返回 100 ÷ n 的结果
{
    if ( n <= 0 )                    // 检查异常：如果 n<=0，则属于异常情况
    {
        exception  err( "输入的人数必须为正整数" );  // 定义 exception 类的对象 err
        throw  ( err );              // 抛出异常对象 err，然后退出当前函数的执行
    }
    // 如果没有异常，则执行如下的正常处理流程
    return  ( 100 / n ) ;
}

int  main( )
{
    int  N;                          // 定义变量 N，保存键盘输入的人数
    cin >> N;
    try
    {
        int  result = Div( N );  // 调用函数 Div 进行除法运算
        cout << "100÷" << N << "=" << result << endl;  // 显示 100 ÷ N 的结果
    }
    catch ( exception  &e )          // 捕捉 exception 类的异常，定义引用参数 e 来接收异常对象
    {
        cout << e.what( ) << endl;        // 向用户显示所捕捉到异常对象 e 的详细信息
    }
    return 0;
}
```

### 3. 捕捉 C++标准库所抛出的异常

C++标准库其他组件所抛出的异常也都来自于 exception 异常类族。例如，使用运算符 new 动态分配内存，如果分配失败（例如内存不足），则将抛出 bad_alloc 类型的异常。bad_alloc 是标准异常类 exception 的派生类。

例如，可使用 try-catch 语句捕捉运算符 new 所抛出的 bad_alloc 类型异常。

```
#include <iostream>
#include <new>
using namespace std;
int main( )
{
    try
    {
        double *p = new double[9999] ;          // 动态分配一个大数组
        cout << "double[9999]：分配成功" << endl ;
        delete [ ]p ;
    }
    catch ( bad_alloc )  // 分配失败时，运算符 new 将抛出 bad_alloc 类型的异常
    {
        cout << "double[9999]：分配失败" << endl ;
    }
    return 0;
}
```

## 本节习题

1. 下列哪种错误不会影响程序的正常执行？（　　）
   A. 语法错误　　B. 语义错误　　C. 运行时错误　　D. 注释错误
2. 异常处理机制主要解决下列哪种错误？（　　）
   A. 语法错误　　B. 语义错误　　C. 运行时错误　　D. 注释错误
3. 下列哪种情况不属于异常处理机制所处理的范畴？（　　）
   A. 用户操作不当　　B. 运行环境存在差异
   C. 网络连接中断　　D. 系统断电
4. C++语言中负责抛出异常的语句是？（　　）
   A. try 子句　　B. catch 子句　　C. throw 语句　　D. if 语句
5. C++语言中负责捕捉异常的语句是？（　　）
   A. try 子句　　B. catch 子句　　C. throw 语句　　D. if 语句
6. 下列关于异常的描述中，错误的是（　　）。
   A. 异常表达式结果的数据类型被用于区分不同类型的异常
   B. 异常表达式结果的值被用于描述异常的详细信息
   C. 每个 catch 子句通常只负责捕捉并处理一种类型的异常
   D. catch( ... )形式的子句捕捉不到任何一种类型的异常
7. 如果某个函数 fun 只会抛出异常类型 A，则正确的函数原型声明语句是（　　）。
   A. void fun( );　　B. void fun( ) throw ( );
   C. void fun( ) throw ( A );　　D. void fun( ) throw A;
8. 下列关于 C++标准库的描述中，错误的是（　　）。
   A. C++标准库预定义了标准异常类 exception

B. 使用标准异常类 exception 需包含头文件<exception>
C. 程序员可以定义自己的异常类
D. 程序员不能继承 C++标准库中的异常类来定义自己的异常类

## 10.5 数据集合及其处理算法

数据集合，就是一组数据的集合。例如，表 10-1 所示的学生成绩单就是一组关于学生姓名和成绩的数据集合。

**表 10-1 学生成绩单**

| 姓 名 | 成 绩 | 姓 名 | 成 绩 |
|---|---|---|---|
| 张三 | 92 | 王五 | 95 |
| 李四 | 86 | … | … |

关于数据集合，存在如下三个层次。

（1）**数据项**（data item）。数据项是数据集合里的最小单位。例如表 10-1 中第 2 行的姓名“张三”、成绩“92”都分别是一个独立的数据项。

（2）**数据元素**（data element）。数据元素是由多个具有内在关联关系的数据项所组成。例如，表 10-1 中第 2 行“张三，92”、第 3 行“李四，86”都分别构成一个数据元素。

（3）**数据集合**（data set）。数据集合由多个并列的数据元素所组成。例如，表 10-1 就是一个由多个数据元素组成的数据集合。

如果数据集合中各数据元素之间的逻辑关系是有序的，则称该数据集合为**有序数据集合**，否则称为**无序数据集合**。对数据集合的处理通常包括**增加**、**查找**、**修改**或**删除**数据元素，统称为**增查改删**（Create、Read、Update 或 Delete，简称 CRUD）。为了提高查找速度，可将数据集合中的数据元素按照某种规则事先进行**排序**（sort）。

编写一个处理数据集合的计算机程序，要涉及两个方面的内容，一是如何组织和存储数据集合，即**数据结构**；二是如何基于数据结构进行数据处理，即**算法**。同样的数据集合，存储的数据结构不同，将导致处理的算法也会有所不同。不同的算法适用于不同数据结构。计算机学科中的“数据结构”就是专门研究数据结构与算法关系的课程。“数据结构”课程的研究对象就是数据集合，其内容是如何对数据集合进行组织、存储和处理的一般方法。

本节将简单介绍存储和处理数据集合的基本原理，并具体讲解如何应用 C++标准库来存储和处理数据集合。C++标准库为数据集合提供了常用的数据结构类及处理算法。

### 10.5.1 数据集合的存储和处理

可以定义结构体数组来存储数据集合。例 10-14 给出一个存储和处理表 10-1 学生成绩单的 C++程序。

**例 10-14　一个存储和处理学生成绩单的 C++程序（使用结构体数组）**

```
#include <iostream>
#include <string.h>
using namespace std;

typedef   struct                         // 定义保存学生成绩的结构体 Student
{
      char   name[9] ;
      float   score ;
}   Student ;

int main( )
{
      Student   s[ 3 ] ;                 // 定义保存学生成绩单的结构体数组 s
      strcpy( s[0].name, "张三" );   s[0].score = 92;   // 通过下标访问结构体数组元素
      strcpy( s[1].name, "李四" );   s[1].score = 86;
      strcpy( s[2].name, "王五" );   s[2].score = 95;

      Student   *p = s ;                 // 通过指针变量 p 访问结构体数组元素
      for   (int n=0; n < 3; n++)        // 显示学生成绩单
      {
            cout << p->name << ", " << p->score << endl;
            p++ ;                        // 将指针变量 p 移到下一个数组元素
      }
      return 0;
}
```

例 10-14 使用结构体数组存储数据集合。定义数组需预先指定元素个数，一次性为所有元素分配一个连续的存储空间。**数组**是存储数据集合时常用的一种基本存储结构。另一种存储数据集合的基本存储结构是**链表**。链表使用动态内存分配方法，每次只为一个数据元素分配内存空间。链表中的数据元素称为链表的**节点**（node）。因为节点各自独立分配内存，所以一个链表中的各个节点之间不一定是连续的，可能分散在内存的不同位置。为了表示不同节点的先后次序，每个节点须增加指向其前一个或后一个节点的指针变量。图 10-3 给出了用链表形式存储表 10-1 学生成绩单的存储示意图。

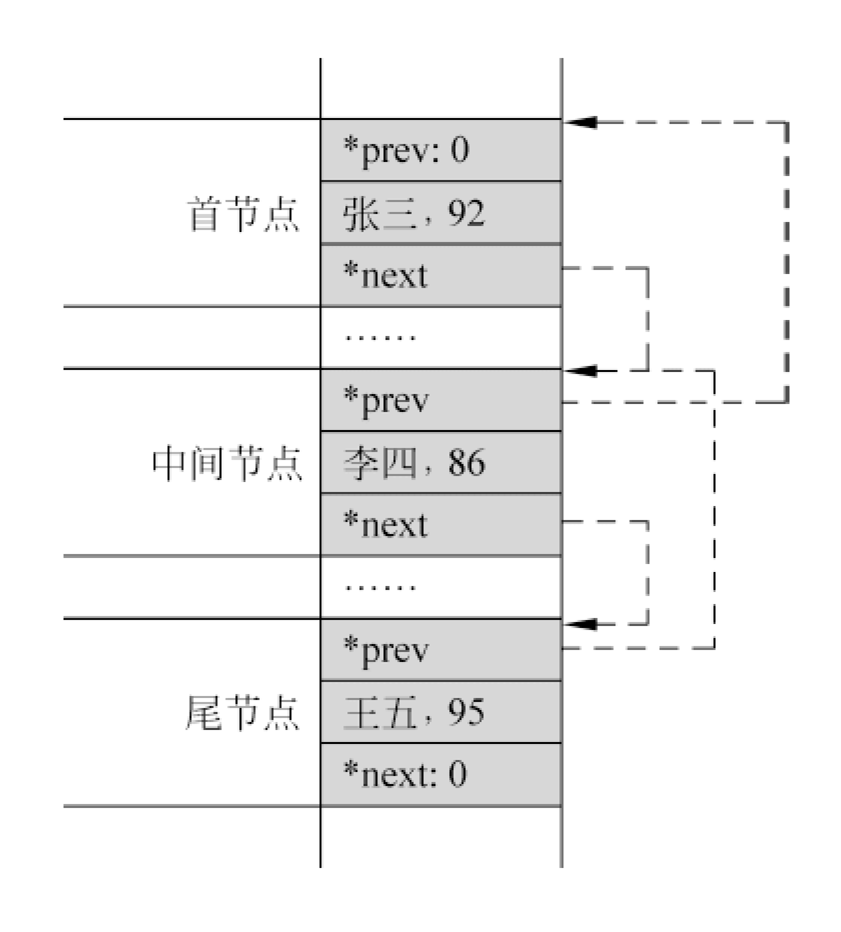



图 10-3　用链表形式存储表 10-1 学生成绩单的存储示意图

链表中的第一个节点被称为**首节点**，最后一个节点被称为**尾节点**。图 10-3 中的每个节点都各增加了一个前向指针 prev 和一个后向指针 next。通过前向/后向指针，程序可以从一个节点查找到其前一个或后一个节点，进而遍历整个链表。首节点是链表的第一个节点，其前向指针为空（0）。尾节点是链表的最后一个节点，其后向指针也为空（0）。同时具有前向和后向指针的链表称为**双**

**向链表**，只有一个方向指针的链表称为**单向链表**。

修改例 10-14 的程序，改用双向链表来存储和处理学生成绩单，其示意代码如例 10-15 所示。

**例 10-15 一个存储和处理学生成绩单的 C++程序（使用双向链表）**

```
#include <iostream>
#include <string.h>
using namespace std;

typedef  struct tagStudent   // 定义保存学生成绩的结构体 Student，具有双向指针
{
    char  name[9] ;
    float  score ;
    struct tagStudent   *prev, *next;  // 定义前向指针 prev 和后向指针 next
}  Student ;

int main( )
{
    Student   *begin ;  // 定义指向双向链表首节点的指针变量 begin
    begin = new   Student;  // 创建首节点
    strcpy( begin->name, "张三" );      begin->score = 92;
    begin->prev = 0;      begin->next = 0;   // 首节点目前还没有前后节点，将前后指针置空
    Student   *curNode = begin;   // 定义指向当前节点的指针变量 curNode

    Student   *p = new   Student;   // 创建第 2 个节点 p
    strcpy( p->name, "李四" );      p->score = 86;
    curNode->next = p;   // 将前一个节点的后向指针 next 指向新节点 p
    p->prev = curNode;   // 将新节点 p 的前向指针 prev 指向前一个节点
    p->next = 0;   // 将新节点 p 的后向指针 next 置空
    curNode = p;   // 新节点 p 成为当前节点

    p = new   Student;   // 创建第 3 个节点 p
    strcpy( p->name, "王五" );      p->score = 95;
    curNode->next = p;   // 将前一个节点的后向指针 next 指向新节点 p
    p->prev = curNode;   // 将新节点 p 的前向指针 prev 指向前一个节点
    p->next = 0;   // 将新节点 p 的后向指针 next 置空
    curNode = p;   // 新节点 p 成为当前节点

    curNode = begin ;   // 通过指针变量 curNode 遍历链表中的各个节点，显示学生成绩单
    while (curNode != 0 )   // 循环条件：当前节点不为空
    {
        cout << curNode ->name << ", " << curNode ->score << endl;
        curNode = curNode->next ;   // 将指针变量 curNode 移到下一个节点
    }
    return 0;
}
```

数组和链表是存储数据集合的两种基本存储结构。与数组相比，链表具有如下三个特点。

（1）**链表便于插入和删除操作**。数组中的数据元素是连续存储的，插入或删除数据元素需要移动该元素后面的所有元素；而链表中的数据元素是各自独立存储的，插入或删除数据元素只需要修改相关的前向和后向指针，不需要移动数据。换句话说，如果数据集合在生成之后比较稳定，不需要经常插入或删除数据元素，则选用数组存储比较适合，否则选用链表更加适合。

（2）**链表只能顺序访问**。数组可以通过下标随机访问任意指定位置的数据元素；而链表只能从头到尾，或从尾到头按顺序依次访问数据元素。

（3）**链表占用内存空间更多**。数组只存储数据元素，而链表除数据外还需要存储前向或后向指针。

## 10.5.2　C++标准库中数据集合的存储和处理

为了存储数据集合，C++标准库提供了 7 个数据结构类，它们被统称为**容器**（container）类。比较常用的容器类有向量类 vector、列表类 list、集合类 set 和映射类 map 等。为了存储不同类型的数据，C++标准库将容器类都定义成了类模板。例如，使用向量类模板 vector 可以定义 int 型向量或 double 型向量，也可以定义结构体类型或类类型的向量。

**迭代器**（iterator）为访问不同容器中的数据元素提供了一种统一的访问方法。每个容器类都定义了自己的迭代器类型 iterator，它是一种类似于指针的类类型，用于保存容器中数据元素的位置信息（相当于地址）。C++标准库按照访问能力的强弱将迭代器划分成 5 种类型，例如某些迭代器只能读取容器中的数据元素，而另一些迭代器则只能向容器中写入数据元素；某些迭代器可以通过下标随机访问容器中任意位置的数据元素，而另一些迭代器则只能通过前向/后向指针按顺序单步访问数据元素（术语称为**单步迭代**）。

为了处理存储在容器中的数据集合，C++标准库提供 70 个左右的函数，具有丰富的数据处理功能，它们被统称为**算法**（algorithm）。算法函数通过迭代器形参接收容器中需要处理的数据元素范围，然后对这些数据元素进行处理。为了接收不同类型的迭代器，C++标准库将算法函数都定义成了函数模板。这样，算法函数就可以处理不同容器类型的数据集合，实现了算法代码的重用。请记住：迭代器非常类似于指针，算法函数通过迭代器访问容器中的数据元素。

本节，我们先介绍 C++标准库中的迭代器和算法。

### 1. 迭代器类型

C++标准库中，迭代器是一种功能类似于指针的类类型，它描述了容器中数据元素的位置信息。如果迭代器对象 p 存放了某个数据元素的位置信息（我们称迭代器对象 p 指向了该数据元素），那么可使用取内容运算符“*”来访问该数据元素，例如“*p”。如果数据元素是某种对象，那么可以使用指向运算符“->”访问其成员，例如“p->成员名”。可以通过自增、自减运算使迭代器指向后一个或前一个数据元素，例如 p++、p--。迭代器类型在理解和使用上都与指针类型非常相似。

在 C++标准库内部，迭代器是用类来实现的。不同迭代器类重载了不同的运算符，因而具有不同的访问能力。C++标准库将迭代器划分成如下的 5 种类型。

**1）输入迭代器（input iterator）**

重载的运算符：++（前置/后置）、*（读内容）、=（赋值）、==（等于）、!=（不等于）。

访问能力：因为只有自增运算，输入迭代器只能按从头到尾的顺序单向访问容器中的数据元素。通过输入迭代器访问数据元素时只能读，不能改。另外，可以比较两个输入迭代器对象是否相等。

**2）输出迭代器（output iterator）**

重载的运算符：++（前置/后置）、*（写内容）、=（赋值）。

访问能力：因为只有自增运算，输出迭代器也只能按从头到尾的顺序单向访问容器中的数据元素。通过输出迭代器访问数据元素时只能写，不能读。

**3）正向迭代器（forward iterator）**

正向迭代器既有输入迭代器的功能，又有输出迭代器的功能。通过正向迭代器访问数据元素时既能读，也能改。同样，正向迭代器也只能按从头到尾的顺序单向访问容器中的数据元素。

**4）双向迭代器（bidirectional iterator）**

双向迭代器在正向迭代器的基础上又重载了自减运算符“– –”，因此双向迭代器既能按从头到尾的顺序，也能按从尾到头的顺序双向访问容器中的数据元素。

**5）随机访问迭代器（random access iterator）**

在双向迭代器的基础上，又重载了如下的运算符：[ ]、+、–、+=、–=、>、>=、<、<=。

访问能力：因为重载了下标运算符“[ ]”，随机访问迭代器可以按下标直接访问容器中任意位置的数据元素。另外，也可以对随机访问迭代器进行加减运算和关系运算。

上述 5 种迭代器类型，从输入迭代器到随机访问迭代器，其访问能力越来越强。不同容器具有不同的迭代器类型，例如向量 vector 的迭代器是随机访问迭代器，而链表 list 的迭代器是双向迭代器。

### 2. 算法

对数据集合的处理通常包括增加、删除、修改或查找数据元素。为了提高查找速度，可将数据集合中的数据元素按照某种规则事先进行排序。C++标准库将算法分成 3 大类，分别为非可变序列算法（即查找算法，见表 10-2）、可变序列算法（即增删改算法，见表 10-3）和排序相关的算法（见表 10-4）。使用 C++标准库的算法需包含其函数声明头文件<algorithm>。

表 10-2 非可变序列算法（12 个）

| | | |
|---|---|---|
| 循环 | for_each | 对序列中的每个元素执行相同的操作 |
| 查找 | find | 在序列中查找某个值第一次出现的位置 |
| | find_if | 在序列中查找符合条件的第一个元素 |
| | find_end | 在序列中查找某个子序列最后一次出现的位置 |
| | find_first_of | 在序列中查找与另一序列第一次出现交集的位置 |
| | adjacent_find | 在序列中找出相邻的重复元素 |
| 计数 | count | 在序列中统计某个值出现的次数 |
| | count_if | 在序列中统计符合条件的元素个数 |

续表

| | | |
|---|---|---|
| 比较 | mismatch<br>equal | 找出两个序列出现不同元素的位置<br>比较两个序列是否相等 |
| 搜索 | search<br>search_n | 在序列中查找某一子序列第一次出现的位置<br>在序列中查找第一个连续出现 $n$ 次指定值的位置 |

**表 10-3　可变序列算法（27 个）**

| | | |
|---|---|---|
| 赋值 | copy<br>copy_backward | 复制序列中指定区间的所有元素<br>逆向复制序列中指定区间的所有元素 |
| 交换 | swap<br>swap_ranges<br>iter_swap | 交换两个元素<br>交换两个序列中的元素<br>交换由迭代器所指向的两个元素 |
| 变换 | transform | 对序列进行函数变换生成新的序列 |
| 替换 | replace<br>replace_if<br>replace_copy<br>replace_copy_if | 将序列中的某个值替换成另一个值<br>将序列中满足条件的元素替换成特定的值<br>复制序列时，将序列中的某个值替换成另一个值<br>复制序列时，将序列中满足条件的元素替换成特定的值 |
| 填充 | fill<br>fill_n | 将序列中指定区间的所有元素填充成某一特定的值<br>将序列中指定区间的前 $n$ 个元素填充成某一特定的值 |
| 生成 | generate<br>generate_n | 对序列指定区间的所有元素做函数变换，用变换结果替换原来的元素<br>对序列指定区间的前 $n$ 个元素做函数变换，用变换结果替换原来的元素 |
| 删除 | remove<br>remove_if<br>remove_copy<br>remove_copy_if | 删除序列中具有给定值的元素<br>删除序列中满足特定条件的元素<br>复制序列时，删除序列中具有给定值的元素<br>复制序列时，删除序列中满足特定条件的元素 |
| 去重 | unique<br>unique_copy | 删除序列中相邻的重复元素<br>复制序列时，删除序列中相邻的重复元素 |
| 倒序 | reverse<br>reverse_copy | 将序列中指定区间元素的次序反转<br>复制序列时，反转序列中元素的次序 |
| 环移 | rotate<br>rotate_copy | 循环移动序列中指定区间的元素<br>复制序列时，循环移动序列中的元素 |
| 洗牌 | random_shuffle | 随机重排序列中指定区间的元素次序 |
| 划分 | partition<br>stable_ partition | 将满足特定条件的元素移到序列的前面<br>将满足特定条件的元素移到序列的前面，并保持原来的相对次序 |

**表 10-4　排序相关算法（27 个）**

| | | |
|---|---|---|
| 排序 | sort<br>stable_sort<br>partial_sort<br>partial_sort_copy | 对序列中指定区间的元素进行排序<br>对序列指定区间的元素进行排序，并保持等值元素原来的相对次序<br>对序列中指定区间的元素进行局部排序<br>复制序列时，对序列中指定区间的元素进行局部排序 |
| 第 $n$ 个元素 | nth_element | 使第 $n$ 个元素符合排序后的次序 |

续表

| | | |
|---|---|---|
| 二分法查找 | lower_bound<br>upper_bound<br>equal_range<br>binary_search | 按升序查找某个特定值应当插入的位置<br>按降序查找某个特定值应当插入的位置<br>查找有序序列中某个特定值可以插入的区间<br>查找有序序列中是否存在与特定值相等的元素 |
| 合并 | merge<br>inplace_merge | 将两个有序序列合并成一个新的序列<br>合并两个相邻的有序序列 |
| 序列的集合运算 | includes<br>set_union<br>set_intersection<br>set_difference<br>set_symmetric_difference | 检查一个序列是否为另一个序列的子序列<br>生成两个集合的有序并集<br>生成两个集合的有序交集<br>生成两个集合的有序差集<br>生成两个集合的有序对称差集(并-交) |
| 堆操作 | push_heap<br>pop_heap<br>make_heap<br>sort_heap | 向堆中压入一个元素<br>从堆中弹出一个元素<br>从序列生成一个堆<br>对堆中元素进行排序 |
| 最大值与最小值 | min<br>max<br>min_element<br>max_element | 返回最小值元素<br>返回最大值元素<br>返回最小值元素的位置<br>返回最大值元素的位置 |
| 字典序比较 | lexicographical_compare<br>next_permutation<br>prev_permutation | 按字典序比较两个序列<br>将序列变换为字典序的下一个排列<br>将序列变换为字典序的前一个排列 |

算法函数通过迭代器形参接收数据元素的位置信息（相当于地址），再通过位置信息（相当于通过地址）访问容器中的数据元素。为了接收不同类型的迭代器实参，C++标准库将算法函数都定义成了函数模板。例如排序函数 sort，其函数原型如下：

```
template   <typename RandomAccessIterator>
void sort( RandomAccessIterator first, RandomAccessIterator last );
```

或：

```
template   <typename RandomAccessIterator>
void sort( RandomAccessIterator first, RandomAccessIterator last, Compare comp );
```

排序函数 sort 被定义成函数模板，其形参可接收不同容器类的迭代器实参，但必需是随机访问类型的迭代器。排序函数 sort 有两种重载形式，形参 first 和 last 分别指定容器中排序区间的首尾元素，而形参 comp 则是以比较函数的形式来指定排序规则。

### 10.5.3　向量类 vector

向量类 vector 是一种容器类，可以存储一维有序数据序列。向量类具有如下特点：

- 向量的内部存储结构是数组。
- 向量中的所有元素都属于同一个数据类型。
- 向量中可以存储的元素数量不受限制。当元素个数超出向量的存储容量时，向量会

自动扩展其内存空间。扩展时会额外多分配若干个元素的存储单元，这样可以避免频繁地扩展内存。由此带来的后果是，向量的后面几个元素可能是空元素，即未使用的元素（会浪费一点内存空间）。

- 向量的迭代器类型是随机访问迭代器，可使用下标访问元素。
- 向量比数组功能更强，可取代数组。

表 10-5 列出了向量类 vector 的简单使用说明。使用向量类需包含其对应的类声明头文件<vector>。

**表 10-5　向量类 vector 的使用说明**

| 使用语法 | 说　　明 |
|---|---|
| vector <class T> v; | 定义一个 T 类型的空向量容器对象 v |
| vector <class T> v( n ); | 定义一个初始元素个数为 n 的 T 类型向量容器对象 v |
| vector <class T>::iterator it; | 定义一个 T 类型向量的迭代器对象 it，该迭代器是随机访问迭代器 |
| v.begin( ) | 返回指向第一个元素位置的迭代器 |
| v.end( ) | 返回指向最后一个元素后面那个空元素位置的迭代器 |
| v.size( ) | 返回向量中元素的个数 |
| v.push_back( e ) | 在向量末尾添加一个元素 e，e 的类型应为 T |
| v.pop_back( ) | 删除向量的最后一个元素 |
| v.insert( it, e ) | 在 it 所指向元素之前插入一个新元素 e，e 的类型应为 T |
| v.erase( it ) | 删除 it 所指向的元素，返回指向其下一个元素位置的迭代器 |
| v.clear( ) | 删除向量中的所有元素 |

向量类 vector 是一个类模板，可使用该类模板定义出不同数据类型的向量对象。下面通过应用举例，分别介绍基本数据类型向量、结构体类型向量和类类型向量。

### 1. 应用举例——基本数据类型向量

例 10-16 以 int 型为例给出一个基本数据类型向量的 C++演示程序。

**例 10-16　一个向量类 vector 的 C++演示程序（int 型向量）**

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector <int>   iv;                    // 定义一个 int 型向量 iv
    for (int n=0; n < 5; n++)             // 添加 5 个向量元素
        iv.push_back( n*n );              // 第 n 个元素的值等于 n 的平方

    vector <int> :: iterator   vit;                      // 定义一个 int 型向量类的迭代器 vit
    for ( vit = iv.begin( ); vit < iv.end( ); vit++ )    // 通过迭代器 vit 遍历向量
        cout << *vit << ", ";                            // 显示向量内容
```

```
    cout << endl << endl;                          // 向量显示结果：0, 1, 4, 9, 16,
    cout << "size= " << iv.size( ) << endl;        // 显示向量中已存放数据的元素个数：size= 5

    iv.pop_back( ) ;                               // 删除向量中的最后一个元素
    cout << "size= " << iv.size( ) << endl;   // 再显示向量中已存放数据的元素个数：size= 4
    return 0;
}
```

例 10-17 给出一个对 int 型向量进行排序的 C++演示程序。

**例 10-17　一个对 int 型向量进行排序的 C++演示程序**

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main( )
{
    vector <int>   iv;              // 定义一个 int 型向量 iv
    // 添加 4 个向量元素，数值依次为：3, 7, 9, 5
    iv.push_back( 3 );      iv.push_back( 7 );      iv.push_back( 9 );      iv.push_back( 5 );

    sort( iv.begin( ), iv.end( ) ); // 使用算法函数 sort 对向量 iv 进行排序，排序区间为整个向量

    // 显示排序后的结果
    vector <int> :: iterator   vit;                // 定义一个 int 型向量类的迭代器 vit
    for ( vit = iv.begin( ); vit < iv.end( ); vit++ )  // 通过迭代器 vit 遍历向量
        cout << *vit << ", ";                      // 显示向量内容
    cout << endl;                                  // 向量排序后的显示结果：3, 5, 7, 9,
    return 0;
}
```

### 2. 应用举例——结构体类型向量

针对表 10-1 的学生成绩单，假设将学生成绩定义成一个结构体类型，则可以使用向量类 vector 来定义一个存储学生成绩单的结构体类型向量。具体代码如例 10-18 所示。

**例 10-18　一个学生成绩单向量的 C++演示程序（结构体类型向量）**

```
#include <iostream>
#include <string.h>
#include <vector>
#include <algorithm>
using namespace std;

typedef   struct   // 定义保存学生成绩的结构体 Student
{
    char   name[9] ;
    float   score ;
}   Student ;
```

```
bool compStudent( Student x, Student y)   // 比较函数：按成绩比较大小。用于指定排序规则
{
    if (x.score < y.score)   return true;      // 此处返回值为 true 则为升序，反之则为降序
    else   return false;
}

int main( )
{
    vector <Student>   sv;   // 定义一个结构体 Student 类型的向量 sv
    // 添加 3 个向量元素
    Student   s;
    strcpy( s.name,  "张三");      s.score = 92;      sv.push_back( s );
    strcpy( s.name,  "李四");      s.score = 86;      sv.push_back( s );
    strcpy( s.name,  "王五");      s.score = 95;      sv.push_back( s );

    sort( sv.begin( ), sv.end( ), compStudent );   // 向量排序，比较函数为 compStudent

    // 显示排序后的结果
    vector <Student> :: iterator   svit;   // 定义一个 Student 类型向量的迭代器 svit
    for ( svit = sv.begin( ); svit < sv.end( ); svit++ )   // 通过迭代器 svit 遍历向量
        cout << svit->name << ", " << svit->score << endl;   // 显示向量内容
    return 0;
}
```

执行例 10-18 的程序，将显示按成绩排序（升序）后的学生成绩单：

```
李四, 86
张三, 92
王五, 95
```

如想要按姓名排序，则需将比较函数 compStudent 修改成如下的形式：

```
bool compStudent( Student x, Student y)   // 比较函数：按姓名比较大小
{
    if (strcmp(x.name, y.name) < 0)   return true;   // 按字符串规则来比较姓名
    else   return false;
}
```

执行修改后的程序，将显示按姓名排序（升序）后的学生成绩单：

```
李四, 86
王五, 95
张三, 92
```

### 3. 应用举例——类类型向量

针对表 10-1 的学生成绩单，假设将学生成绩定义成一个类，则可以使用向量类 vector 来定义一个存储学生成绩单的类类型向量。具体代码如例 10-19 所示。

例 10-19　一个学生成绩单向量的 C++演示程序（类类型向量）

```
#include <iostream>
#include <string.h>
#include <vector>
#include <algorithm>
using namespace std;

class   Student   // 定义保存学生成绩的类 Student
{
public:
      char   name[9] ;
      float   score ;
      bool operator<(Student x)   // 重载运算符<：按成绩比较大小。其用途是指定排序规则
      {
            if (score < x.score)   return true;   //此处返回值为 true 则为升序，反之则为降序
            else   return false;
      }
} ;

int main( )
{
      vector <Student>   sv;   // 定义一个类 Student 类型的向量 sv
      // 添加 3 个向量元素
      Student   s;
      strcpy( s.name,   "张三");      s.score = 92;      sv.push_back( s );
      strcpy( s.name,   "李四");      s.score = 86;      sv.push_back( s );
      strcpy( s.name,   "王五");      s.score = 95;      sv.push_back( s );

      sort( sv.begin( ), sv.end( ) );   // 对向量 sv 进行排序。排序时将使用重载的运算符“<”

      // 显示排序后的结果
      vector <Student> :: iterator   svit;   // 定义一个类 Student 类型向量的迭代器 svit
      for ( svit = sv.begin( ); svit < sv.end( ); svit++ )   // 通过迭代器 svit 遍历向量
            cout << svit->name << ", " << svit->score << endl;   // 显示向量内容
      return 0;
}
```

执行例 10-19 的程序，将显示按成绩排序（升序）后的学生成绩单：

```
李四, 86
张三, 92
王五, 95
```

## 10.5.4　列表类 list

列表类 list 是一种容器类，可以存储一维有序数据序列。列表中的所有元素都属于同一个数据类型。与向量类 vector 相比，列表类 list 具有如下特点。

■ 列表的内部存储结构是链表，而向量的内部存储结构是数组。

■ 列表中每个元素的内存空间是独立分配的，而向量是连续分配的。
■ 列表适合于存储需要频繁添加、删除的数据集合，而向量适合于存储元素总数相对固定的数据集合，即向量不适合频繁地添加或删除元素。
■ 列表的迭代器类型是双向迭代器（不能使用下标访问元素），而向量的迭代器类型是随机访问迭代器（可以使用下标访问元素）。

从使用角度看，列表类 list 的使用方法与向量类 vector 比较类似。使用列表类需包含其对应的类声明头文件<list>。列表类 list 是一个类模板，可使用该类模板定义出不同数据类型的列表对象。例 10-20 给出一个 int 型列表的 C++演示程序。

**例 10-20　一个列表类 list 的 C++演示程序（int 型列表）**

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main( )
{
    list <int>   il;        // 定义一个 int 型列表 il
    // 添加 4 个列表元素，数值依次为：3, 7, 9, 5
    il.push_back( 3 );    il.push_back( 7 );    il.push_back( 9 );    il.push_back( 5 );

    il.sort( );             // 对列表 il 进行排序。列表类 list 自带排序函数成员 sort

    // 显示排序后的结果
    list <int> :: iterator   lit;    // 定义一个 int 型列表类的迭代器 lit（属于双向迭代器）
    for ( lit = il.begin( ); lit != il.end( ); lit++ )   // 通过迭代器 lit 遍历列表
        cout << *lit << ", ";        // 显示列表内容
    cout << endl;                    // 列表排序后的显示结果：3, 5, 7, 9,
    return 0;
}
```

## 10.5.5　集合类 set

集合类 set 是一种容器类，可以存储一维无序数据序列。集合类 set 具有如下特点。

■ 集合中的所有元素都属于同一个数据类型。
■ 向集合中插入的元素总是会按某种键值被自动排序，可通过比较函数来指定排序键值。排序的目的是为了今后快速查找集合中的元素。
■ 集合中每个元素的键值都是唯一的，不能相同。插入键值重复的元素时，新元素将覆盖老元素。
■ 集合的迭代器类型是双向迭代器（因为集合的内部存储结构是双向链表）。

集合类 set 是一个类模板，可使用该类模板定义出不同数据类型的集合对象。使用集合类需包含其对应的类声明头文件<set>。例 10-21 给出一个 int 型集合的 C++演示程序。

例 10-21　一个集合类 set 的 C++演示程序（int 型集合）

```
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;

int main( )
{
    set <int>   is;   // 定义一个 int 型集合 is
    // 插入 4 个集合元素，数值依次为：3, 7, 9, 5
    is.insert( 3 );     is.insert( 7 );     is.insert( 9 );     is.insert( 5 );

    is.insert( 3 );   // 重复元素：键值重复的元素 3 将自动覆盖之前的老元素 3

    // 显示集合中的元素
    set <int> :: iterator   sit;   // 定义一个 int 型集合类的迭代器 sit（属于双向迭代器）
    for ( sit = is.begin( ); sit != is.end( ); sit++ )   // 通过迭代器 sit 遍历集合
        cout << *sit << ", ";   // 显示集合内容
    cout << endl;                 // 集合会自动排序，显示结果：3, 5, 7, 9,
    return 0;
}
```

## 10.5.6　映射类 map

映射类 map 是一种适合存储被称作“**键-值**”类型数据的容器类。映射类 map 具有如下特点。

- “键-值”类型数据包括两个数据，一个称为“键”，一个称为“值”。例如：“张三, 92”“李四, 86”……就是一种“姓名-分数”类型的键值对，其中姓名是“键”，分数是“值”。
- 向映射中插入的元素总是被按键自动排序，这样可以便于今后的快速查找。
- 映射中每个元素的键都是唯一的，不能相同。插入键重复的元素时，新元素将覆盖老元素。
- 映射的迭代器类型是双向迭代器（因为映射的内部存储结构是双向链表）。

映射类 map 是一个类模板，可使用该类模板定义出不同数据类型的映射对象。使用映射类需包含其对应的类声明头文件<map>。针对表 10-1 的学生成绩单，可以使用映射类 map 来定义一个存储学生成绩单的映射。具体代码如例 10-22 所示。

例 10-22　一个映射类 map 的 C++演示程序

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main( )
{
```

```
    map <string, int>   sim;  // 定义一个 string-int 型映射 sim
    // 插入 3 个映射元素，数值依次为：张三-92，李四-86，王五-95
    sim["张三"] = 92;     sim["李四"] = 86;     sim["王五"] = 95;

    sim["张三"] = 100;  // 重复元素：“键”重复的元素“张三”将自动覆盖之前的老元素

    map <string, int> :: iterator   mit;  // 定义一个 string-int 型映射类的迭代器 mit
    for (mit = sim.begin( ); mit != sim.end( ); mit++)  // 通过迭代器 mit 遍历映射
        cout << mit->first << ", " << mit->second << endl;  // 显示键（first）和值（second）
    return 0;
}
```

执行例 10-22 的程序，将显示按键（姓名）排序后的学生成绩单：

```
李四, 86
王五, 95
张三, 100
```

## 本节习题

1. 不属于数据集合基本存储结构的是（　　）。
   A. 数组　B. 单向链表　C. 双向链表　D. 结构体
2. 与 C++标准库中容器概念关联最小的知识点是（　　）。
   A. 数据存储　B. 类类型　C. 向量类 vector　D. 控制语句
3. 与 C++标准库中迭代器概念关联最小的知识点是（　　）。
   A. 指针类型　B. 重载运算符　C. 随机访问　D. 引用传递
4. 与 C++标准库中算法概念关联最小的知识点是（　　）。
   A. 数据处理　B. 数据排序　C. 数据查找　D. 数据成员
5. 下列定义向量对象的语句中，错误的是（　　）。
   A. vector <int>　x;　B. vector <double>　x;
   C. vector <string>　x;　D. vector　x;
6. 下列定义列表对象的语句中，错误的是（　　）。
   A. list <int>　x;　B. list <double>　x;
   C. list <string>　x;　D. list　x;
7. 下列关于向量类 vector 与列表类 list 的描述中，错误的是（　　）。
   A. 向量和列表的内部存储结构相同，都是链表
   B. 列表中每个元素的内存空间是独立分配的，而向量的内存空间是连续分配的
   C. 列表适合存储要频繁添加、删除的数据集合，而向量不适合频繁地添加、删除元素
   D. 列表的迭代器类型是双向迭代器，而向量的迭代器类型是随机访问迭代器
8. 双向迭代器不能进行下列哪种运算？（　　）
   A. 自增运算++　B. 自减运算--　C. 指针运算*　D. 下标运算[ ]

# 10.6　面对象程序设计总结

通过本书共 10 章的学习，相信读者已基本掌握了程序设计的基本原理和方法，并能够比较熟练地运用 C++语言的语法知识来阅读或编写简单的计算机程序。通过重用他人的代码，例如标准 C 库中的系统函数或 C++标准库中的系统类库，程序员可以站在更高的起点上开发程序，而不需要从零开始。

除了标准 C 库和 C++标准库，还有大量第三方开发的函数库或类库可供程序员选用。例如，使用 Microsoft Visual C++ 6.0 集成开发环境的程序员可以选用微软公司提供的微软基础类库（Microsoft Foundation Classes，MFC）。使用 MFC 类库，程序员可以快速开发出各种功能强大的应用程序。例如：

- **基于 Windows 的图形用户界面程序**。MFC 类库提供了多种窗口及组件类，常用的有窗口类 CWnd、对话框类 CDialog、按钮类 CButton、编辑框类 CEdit 和菜单类 CMenu 等。
- **网络应用程序**。MFC 类库提供了多种网络通信类，常用的有套接字类 CSocket、HTTP 连接类 CHttpConnection 和 FTP 连接类 CFtpConnection 等。
- **数据库应用程序**。MFC 类库提供了统一的数据库操作类，常用的有数据库类 CDaoDatabase 和记录集类 CRecordSet 等。

## 10.6.1　使用 MFC 类库开发图形用户界面程序

下面我们以温度换算程序为例，简单介绍一下如何使用 MFC 类库来开发图形用户界面的 C++程序（如图 10-4 所示）。假设用户在【摄氏温度】后面的编辑框中输入摄氏温度 10，单击【转换】按钮，程序将执行温度转换算法，并将转换得到的华氏温度 50.0 显示在【华氏温度】后面的文本框中。如果用户单击了【退出】按钮，则关闭程序窗口，退出程序。




图 10-4　具有图形用户界面的温度换算程序

如何在 Windows 桌面上画一个窗口，或在窗口中画一个按钮？为了帮助程序员开发 Windows 程序，微软公司运用面向对象程序设计方法，将窗口、编辑框和按钮等界面元素编写成类。程序员用这些类定义对象，再调用对象的 ShowWindow 方法，就可以创建并显示这些界面元素了。如果程序员想修改或扩充界面元素的外观或功能，那么可以使用继承

与派生的方法定义自己的界面元素类（即派生类）。图 10-4 中的程序窗口包含一个【摄氏温度】编辑框、一个【转换】按钮和一个【退出】按钮，可以将这个窗口类定义成一个组合类，它包含了一个编辑框类和两个按钮类的对象成员。MFC 类库随 Visual C++ 6.0 或 Visual Studio 系列集成开发环境提供。可以使用 Visual C++ 6.0 并基于 MFC 中相关的类很容易地编写出一个具有图形用户界面的 Windows 程序。使用 Visual C++ 6.0 编写上述温度换算程序，编程过程需分如下 3 步完成。

**第 1 步**：新建一个基于 MFC 类库的工程（图 10-5）。假设工程名为 mfctest，新建时选择 MFC AppWizard(exe)选项。**注**：之前我们编写命令行界面程序时新建的是 Win32 Console Application 工程（即控制台应用程序）。




图 10-5 新建一个基于 MFC 类库的工程

MFC AppWizard 是一个 MFC 应用程序向导，可帮助程序员快速创建程序。按照 MFC 应用程序向导的提示一步一步进行操作，将应用程序类型设为【基本对话框】（图 10-6），这样就可以新建一个具有对话框风格的图形用户界面程序。



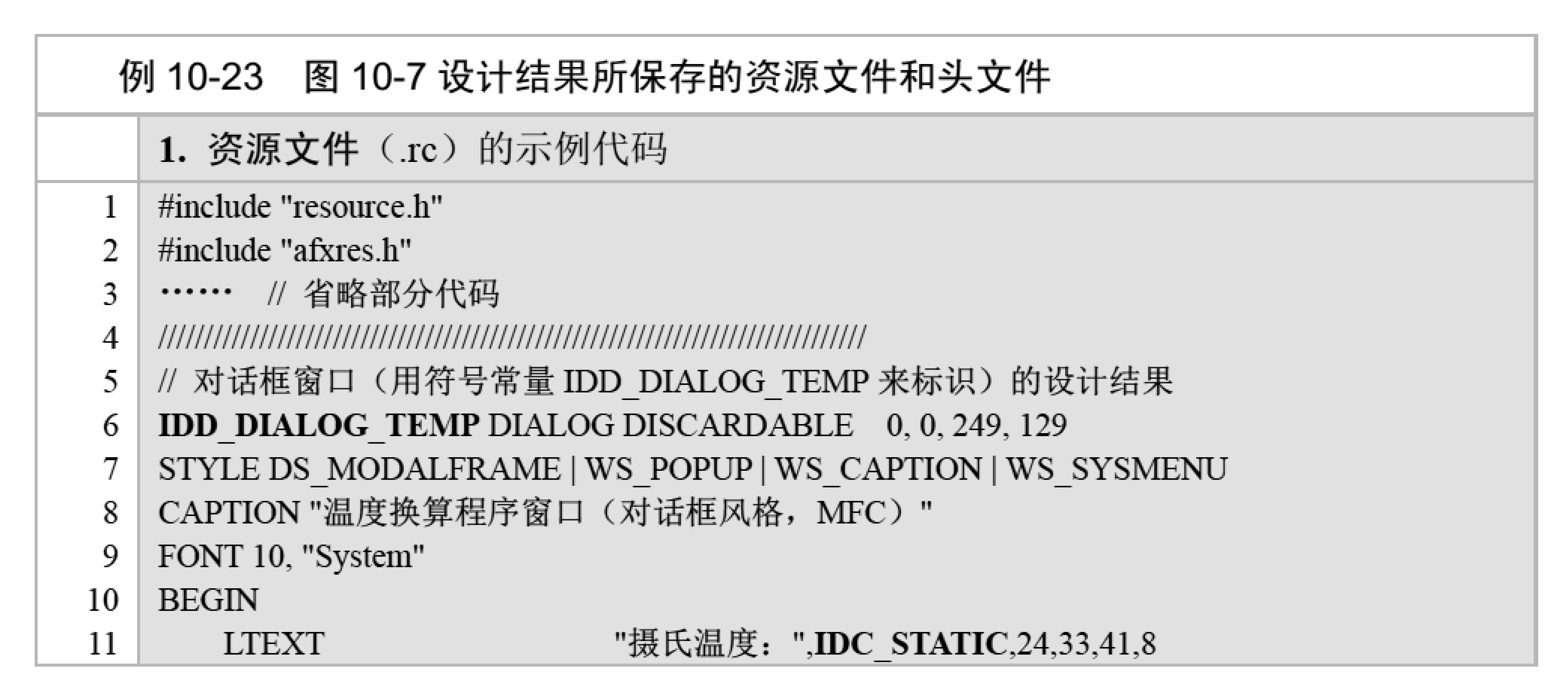

图 10-6 新建一个具有对话框风格的图形用户界面程序

**第 2 步**：使用 Visual C++ 6.0 的图形界面设计器来设计窗口界面（如图 10-7 所示）。图形界面设计器提供很多控件，例如静态文本框、编辑框和按钮等，程序员可以用拖动的方法在窗口中添加这些界面元素。添加界面元素时需为它们分别指定不同的整数编号（用符号常量来表示）。例如，添加一个用于输入摄氏温度的编辑框，并用符号常量 IDC_EDIT_CTEMP 作为其整数编号。




图 10-7　Visual C++ 6.0 的图形界面设计器

图形界面设计器将设计结果保存到一个资源文件（扩展名为.rc）中。Windows 操作系统将图形界面相关的元素统称为**资源**（resource）。图形界面设计器还将各界面元素所对应的符号常量定义在头文件 resource.h 中。图 10-7 设计结果所保存的资源文件和头文件如例 10-23 所示。



**例 10-23　图 10-7 设计结果所保存的资源文件和头文件**

**1. 资源文件**（.rc）的示例代码

```
#include "resource.h"
#include "afxres.h"
……  // 省略部分代码
/////////////////////////////////////////////////////////////////////////////
// 对话框窗口（用符号常量 IDD_DIALOG_TEMP 来标识）的设计结果
IDD_DIALOG_TEMP DIALOG DISCARDABLE   0, 0, 249, 129
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "温度换算程序窗口（对话框风格，MFC）"
FONT 10, "System"
BEGIN
    LTEXT           "摄氏温度：",IDC_STATIC,24,33,41,8
```

```
    EDITTEXT            IDC_EDIT_CTEMP,65,31,125,14,ES_AUTOHSCROLL
    LTEXT               "",IDC_STATIC_FTEMP,25,56,165,8
    PUSHBUTTON          "转换",IDC_BUTTON_CALC,192,30,50,14
    DEFPUSHBUTTON       "退出",IDC_BUTTON_QUIT,192,108,50,14
END
```

**2. 头文件（resource.h）的示例代码**

```
#define IDD_DIALOG_TEMP      101     // 温度换算程序窗口的编号
#define IDC_EDIT_CTEMP       1000    //【摄氏温度】编辑框的编号
#define IDC_STATIC_FTEMP     1001    //【华氏温度】文本框的编号
#define IDC_BUTTON_CALC      1002    //【转换】按钮的编号
#define IDC_BUTTON_QUIT      1003    //【退出】按钮的编号
```

**第 3 步：**编写温度换算程序的完整 C++代码。基于 MFC 的 Windows 图形用户界面程序主要包含 2 个类：一是应用程序类，二是窗口类。MFC 应用程序向导自动为程序员编写好部分代码，并将类的声明代码、实现代码分别保存在头文件（.h）和源程序文件（.cpp）中，非常贴心！本例中的应用程序类是 CMfctestApp，窗口类是 CMfctestDlg（对话框风格的窗口）。例 10-24 给出了应用程序类 CMfctestApp 的示意代码。

例 10-24　应用程序类 CMfctestApp 的示意代码

**1. 类声明文件（mfctest.h）**

```
class CMfctestApp : public CWinApp      // 公有继承 MFC 类 CWinApp
{
public:
    CMfctestApp( );                     // 构造函数
    virtual BOOL InitInstance( );       // 入口函数（是一个虚函数）
     …  // 省略部分代码
};
```

**2. 类实现程序文件（mfctest.cpp）**

```
#include "stdafx.h"                     // MFC 相关的头文件
#include "mfctest.h"                    // 声明类 CMfctestApp 的头文件
#include "mfctestDlg.h"                 // 声明类 CMfctestDlg 的头文件
CMfctestApp::CMfctestApp( )             // 构造函数的定义
{
    // 本例中，构造函数不需要做什么事情
}
BOOL CMfctestApp::InitInstance( )       // 重写入口函数
{
    CMfctestDlg dlg;                    // 定义一个对话框窗口类对象 dlg
    dlg.DoModal( );                     // 显示程序窗口，等待用户操作
    return 0;
}

CMfctestApp theApp;                     // 定义一个应用程序类对象 theApp
```

应用程序类 CMfctestApp 是 MFC 中 CWinApp 的派生类。它封装了主函数，并为程序员新增一个入口函数 InitInstance( )。计算机执行这个程序，将首先执行定义对象语句：

```
CMfctestApp    theApp;                    // 定义一个应用程序类对象 theApp
```

在构造对象 theApp 时调用入口函数 InitInstance( )，这样程序执行流程就进入了程序员可以控制的范围。在 MFC 图形用户界面程序中，入口函数 InitInstance( )承担起了主函数 main( )的职能。例 10-24 中，程序员在入口函数 InitInstance( )中编写如下 2 条语句来创建并显示程序窗口：

```
CMfctestDlg    dlg;                       // 定义一个对话框窗口类对象 dlg
dlg.DoModal( );                           // 显示程序窗口，等待用户操作
```

计算机执行 dlg.DoModal( )函数，将显示程序窗口，等待用户操作。至此，应用程序类 CMfctestApp 的工作就完成了，下面的工作交由对话框窗口类 CMfctestDlg 继续完成。例 10-25 给出了对话框窗口类 CMfctestDlg 的示意代码。

**例 10-25　对话框窗口类 CMfctestDlg 的示意代码**

**1. 类声明文件（CMfctestDlg.h）**

```
class CMfctestDlg : public CDialog                  // 公有继承 MFC 类 CDialog
{
public:
     CMfctestDlg(CWnd* pParent = NULL);       // 构造函数
     enum { IDD = IDD_DIALOG_TEMP }; // 按照资源 IDD_DIALOG_TEMP 来创建对话框窗口
     CEdit    m_eCTemp;                   // 包含一个编辑框类 CEdit 的对象成员
     CStatic   m_sFTemp;                  // 包含一个文本框类 CStatic 的对象成员
protected:
     afx_msg void OnButtonCalc( );        // 单击【转换】按钮时调用该函数
     afx_msg void OnButtonQuit( );        // 单击【退出】按钮时调用该函数
      ……                                // 省略部分代码
};
```

**2. 类实现程序文件（CMfctestDlg.cpp）**

```
#include "stdafx.h"                                  // MFC 相关的头文件
#include "mfctest.h"                                 // 声明类 CMfctestApp 的头文件
#include "mfctestDlg.h"                              // 声明类 CMfctestDlg 的头文件
#include <stdio.h>
CMfctestDlg::CMfctestDlg(CWnd* pParent=NULL)         // 构造函数
              : CDialog(CMfctestDlg::IDD, pParent)   // 初始化列表
{
     // 本例中，构造函数不需要做其他事情
}

BEGIN_MESSAGE_MAP(CMfctestDlg, CDialog)   // 处理用户不同的操作（称为消息映射）
     ON_BN_CLICKED(IDC_BUTTON_CALC, OnButtonCalc) // 单击【转换】按钮时调用 OnButtonCalc
     ON_BN_CLICKED(IDC_BUTTON_QUIT, OnButtonQuit) // 单击【退出】按钮时调用 OnButtonQuit
END_MESSAGE_MAP( )
```

```
15
16 void CMfctestDlg::OnButtonCalc( )   // 处理单击【转换】按钮消息，进行温度转换
17 {
18     char str[50];
19     m_eCTemp.GetWindowText(str, 50);
20     double ctemp;
21     sscanf(str, "%lf", &ctemp);
22     double ftemp = ctemp*1.8 + 32;
23     sprintf(str, "华氏温度：%6.1lf", ftemp);
24     m_sFTemp.SetWindowText(str);
25 }
26 void CMfctestDlg::OnButtonQuit( )   // 处理单击【退出】按钮消息，关闭窗口，退出程序
27 {
28     CDialog::OnOK( );   // 调用基类 CDialog 的函数成员 OnOK( )来关闭窗口，并退出程序
29 }
30 ……   // 省略部分代码
```

例 10-25 中，对话框窗口类 CMfctestDlg 首先继承 MFC 中的标准对话框类 CDialog，然后新增了 2 个数据成员（例如编辑框类 CEdit 的对象成员 m_eCTemp），还新增了 2 个函数成员（例如 OnButtonCalc 函数）。可以看出，定义 CMfctestDlg 类时同时使用了继承和组合的方法，它是一个组合派生类。CMfctestDlg 类新增函数成员的目的是处理用户在窗口中的操作，实现相应的程序功能。

计算机可同时运行多个程序，桌面上将显示多个程序窗口。Windows 操作系统负责监控鼠标动作并捕获用户的操作，根据鼠标位置来判断用户对哪个程序窗口进行了操作。如果用户对某个窗口进行了操作（例如单击了某个按钮），那么 Windows 操作系统将自动调用该窗口对象所指定的处理函数。Windows 操作系统以**消息**（message）的形式来区分用户进行了何种操作。MFC 程序通过**消息映射**来指定各消息所对应的处理函数。例 10-25 中，代码第 11~14 行就是对话框窗口类 CMfctestDlg 的消息映射，它指定单击【转换】按钮时调用消息处理函数 OnButtonCalc，单击【退出】按钮时调用消息处理函数 OnButtonQuit。

用户单击【转换】按钮，计算机将执行消息处理函数 OnButtonCalc( )（代码第 16~25 行），其算法流程是：首先取出【摄氏温度】编辑框（即 IDC_EDIT_CTEMP）中的数据（字符串类型），转成 double 型数值，然后计算其对应的华氏温度，再转成字符串形式并显示在【华氏温度】文本框（即 IDC_STATIC_FTEMP）中。这样，用户输入摄氏温度，单击【转换】按钮，就可以在【华氏温度】文本框中看到换算结果。

用户单击【退出】按钮，计算机将执行消息处理函数 OnButtonQuit( )，通过调用基类 CDialog 的函数成员 OnOK( )来关闭窗口，并退出程序。

上面通过温度换算程序的例子简单介绍了 Windows 图形用户界面程序的开发过程，其中的某些语法细节没有展开，但这不影响对程序的理解。

### 10.6.2 结语

通过上一小节温度换算程序的开发过程可以看出，只要使用 MFC 类库，程序员就可以快速开发出 Windows 图形用户界面程序。以类的形式来组织和重用代码可以极大地提高

程序开发效率，这就是面向对象程序设计思想的精髓所在。

在学习完面向对象程序设计之后，程序员在拿到一个具体的程序设计任务时，首先应当考虑有哪些现成的类库可以用，实在找不到所需要的类库才考虑自己编写程序代码。基于已有的类库开发程序，相当于是用别人已经做好的零件来组装产品。程序的应用开发，通常就是用已有的程序零件来组装自己的软件产品。

只要掌握了面向对象程序设计方法和 C++语言，相信每位读者都能够借助各种第三方类库，发挥出无限的开发潜能。

### 学习本章的要点

- 读者应了解如何使用模板技术来提高函数和类代码的可重用性。
- 读者应重点学习 C++语言的异常处理机制。
- 读者应初步掌握如何使用 C++标准库中的向量类、列表类、集合类和映射类来存储和处理数据集合。

## 10.7　本章习题

1. **编写程序**。模仿例 10-1 设计一个求最小值的函数模板 Min，并编写主函数进行测试。

2. **编写程序**。模仿例 10-10 的代码结构编写一个求两个实数之和平方根（提示：调用系统函数 sqrt）的 C++程序。要求使用 try-catch 异常处理机制处理对负数求平方根的异常。

3. **编写程序**。模仿例 10-16 的代码结构编写一个求学生平均成绩的 C++程序。要求使用 C++标准库中的向量来存储学生成绩。




# 附录A Visual C++ 6.0集成开发环境

Microsoft Visual C++ 6.0，简称 VC6 或 VC 6.0，是美国微软公司推出的集编码、编译、连接和调试于一体的 C++语言集成开发环境(Integrated Development Environment，IDE)。VC 6.0 软件需单独安装，安装后才能使用。图 A-1 给出 VC 6.0 启动之后的主操作界面。

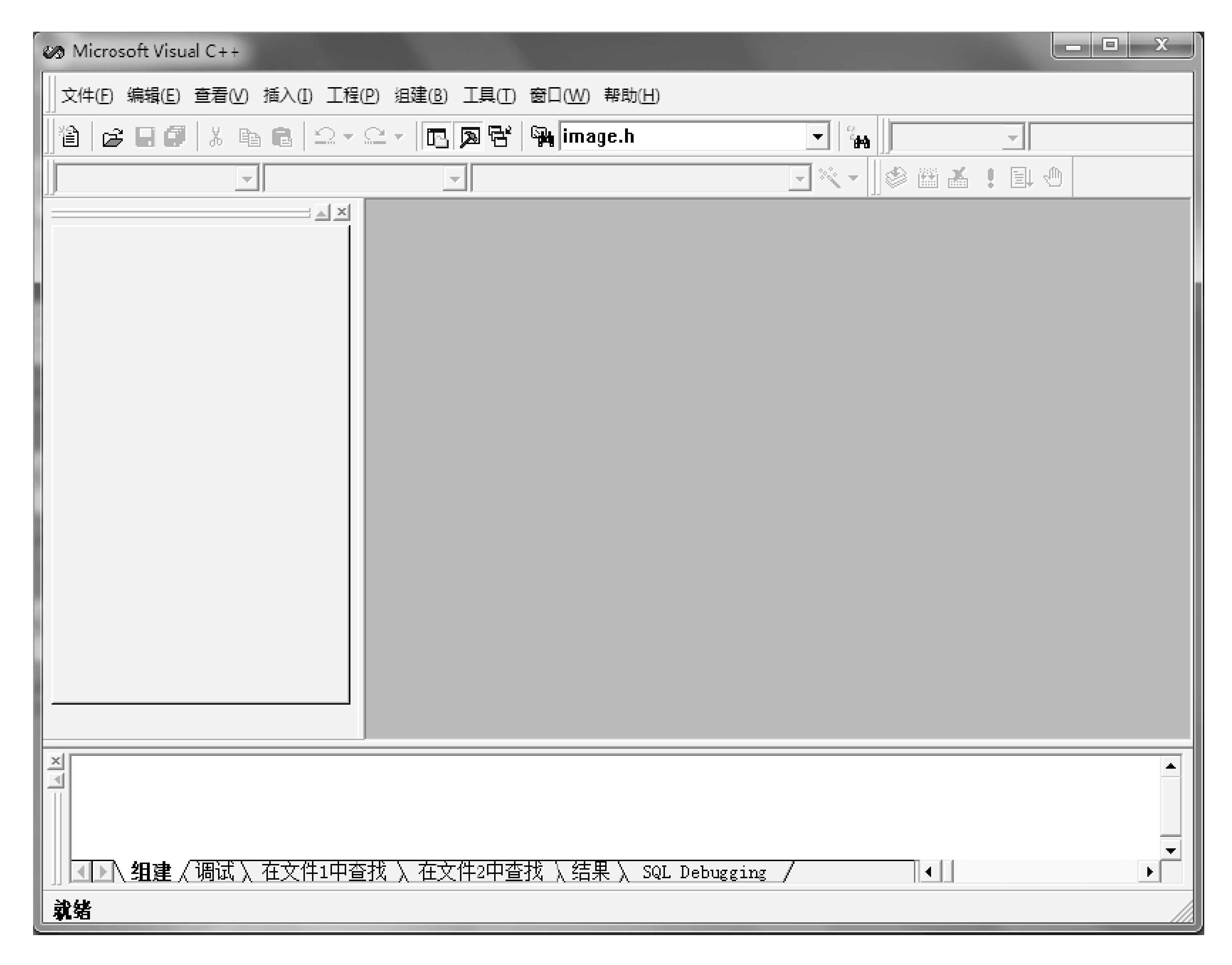



图 A-1　VC 6.0 的主操作界面

作为程序设计的初学者，重要的是学习程序设计的基本原理和方法，培养运用计算思维分析和解决问题的能力。初学者阅读本书时需一边学习语法知识，一边上机练习，例如使用 VC 6.0 来编写简单的 C++程序。上机练习是巩固并深入领会程序设计原理和 C++语法知识的最主要手段。

VC 6.0 将命令行界面的程序称作控制台应用(console application)程序。使用 VC 6.0 编写一个控制台应用程序，程序员需按如下步骤操作。

1. 新建工程

单击菜单【文件】→【新建】，进入图 A-2 所示的【新建】对话框。

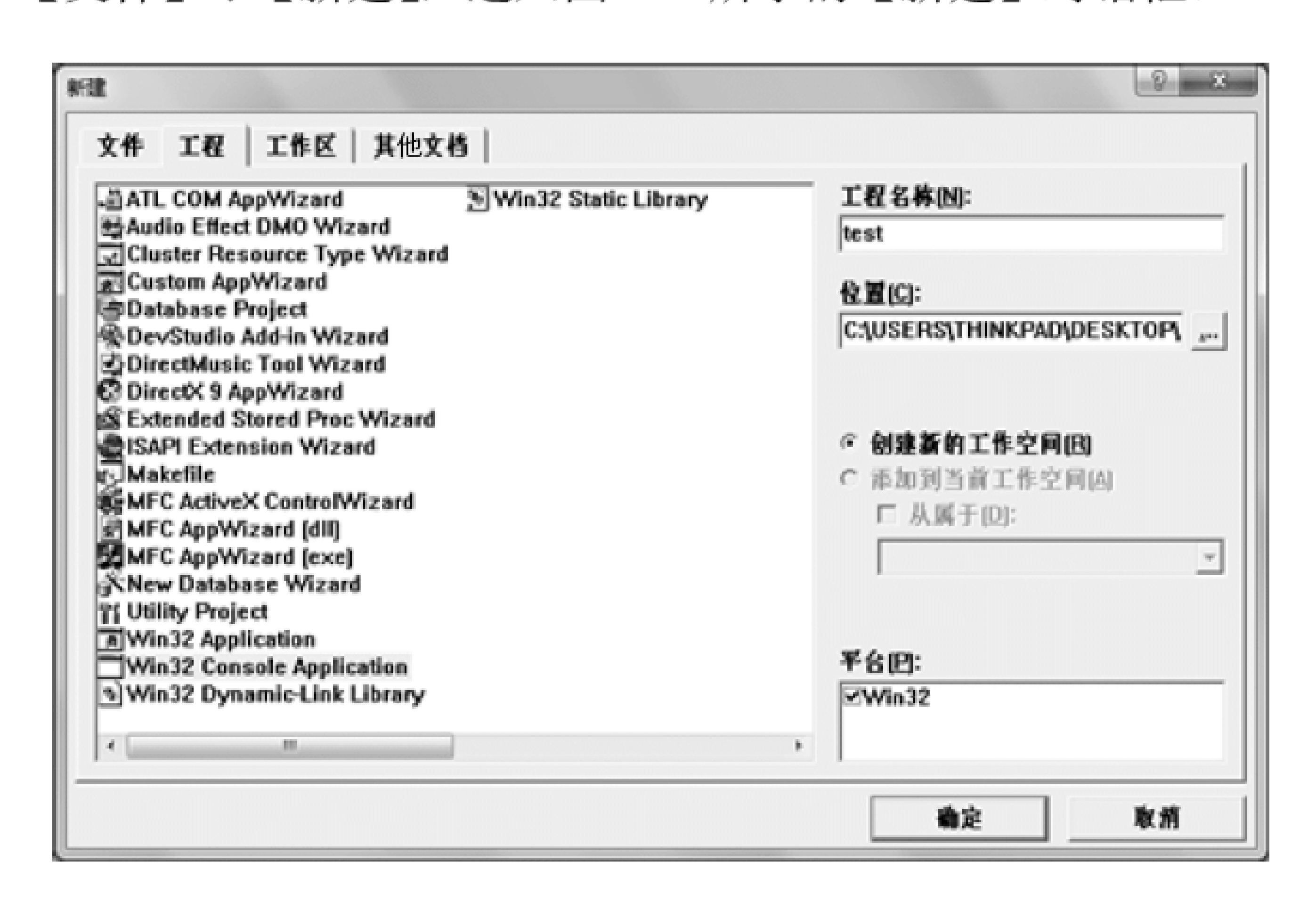



图 A-2　【新建】对话框

在图 A-2 所示的新建对话框中，首先在左侧选择 Win32 Console Application 工程，将新建工程的类型设置为控制台应用程序；然后在右侧【工程名称】输入工程的名称，在【目录】输入保存该工程的目录位置。在 VC 6.0 中，每新建一个工程都将创建一个工程目录，该目录的名字为工程名称。今后工程中所有的源程序文件及所生成的目标文件和可执行文件等都将保存到这个工程目录。本例是在桌面上新建一个名为 test 的控制台应用程序工程。单击【确定】按钮，继续进入图 A-3 所示的步骤 1 对话框。

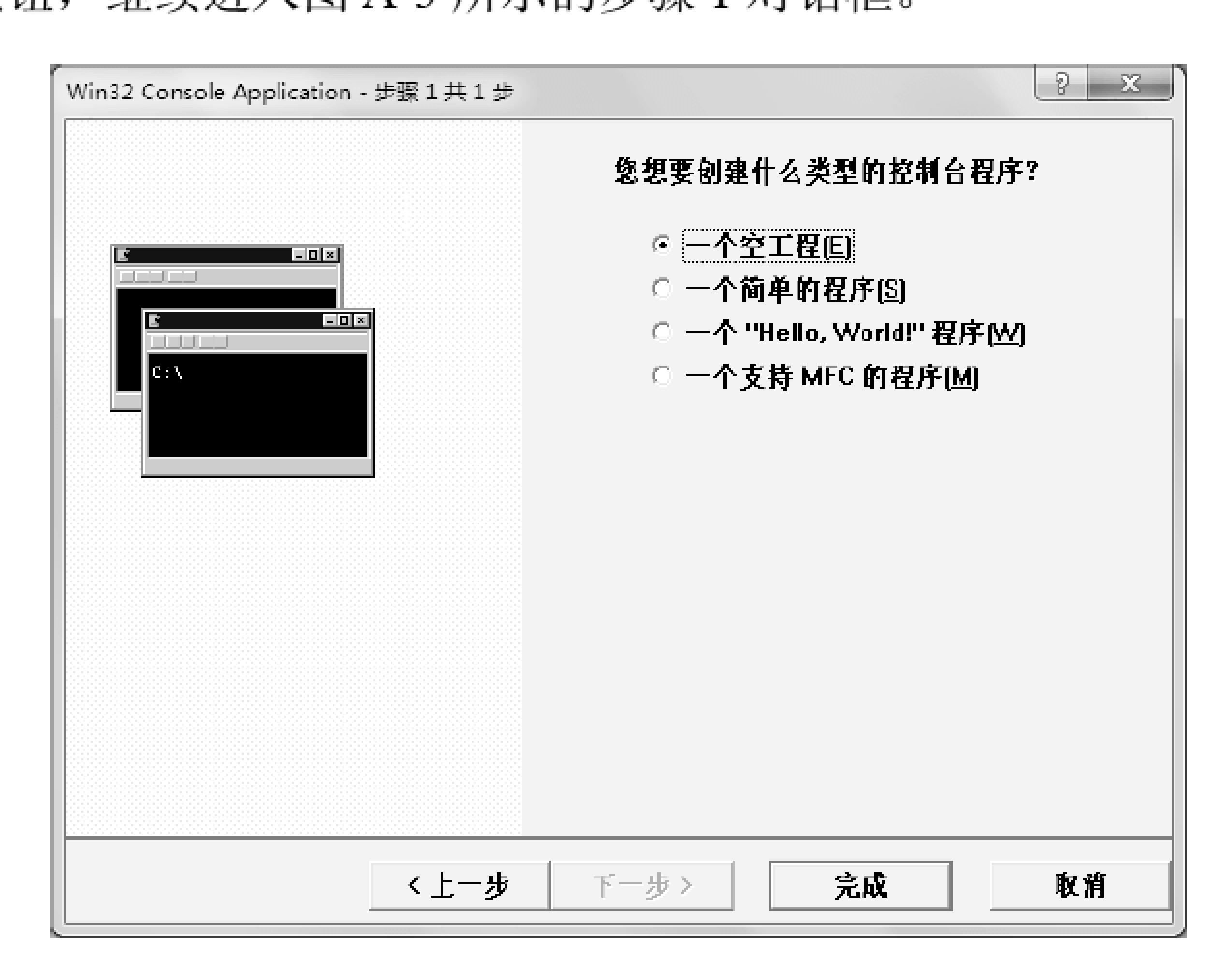



图 A-3　步骤 1 对话框

在图 A-3 所示的步骤 1 对话框中，选择创建【一个空工程】，然后单击【完成】按钮，完成新建工程的操作。

2. 新建 C++源程序文件

再次单击菜单【文件】→【新建】，进入图 A-4 所示的【新建】对话框。




图 A-4 【新建】对话框

在图 A-4 所示的【新建】对话框中，首先在左侧选择 C++ Source File 文件，将新建文件的类型设置为 C++源程序文件；然后在右侧【文件名】下方输入文件名，并将该文件添加到之前新建的工程中(选中【添加到工程】)。本例新建一个名为 1.cpp 的 C++源程序文件，并将其添加到工程 test 中。单击【确定】按钮，继续进入图 A-5 所示的编辑源程序界面，输入 C++源程序。本例输入了一个将摄氏温度换算成华氏温度的 C++程序。

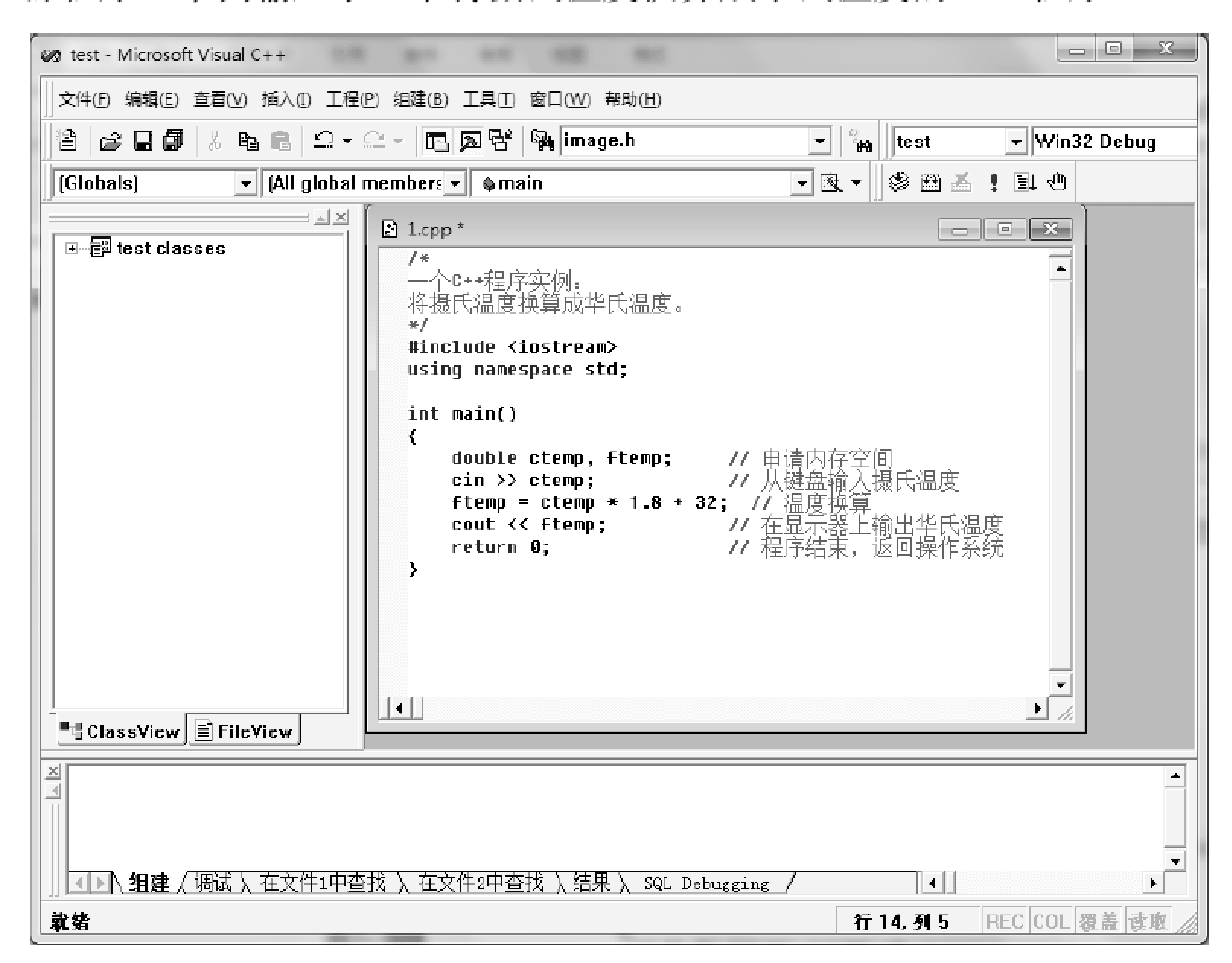



图 A-5 编辑源程序界面

### 3. 组建程序

单击菜单【组建】→【组建】，VC 6.0 开始对工程中的 C++源程序文件进行编译、连接。如果没有语法错误，将生成可执行的程序文件。该可执行程序的文件名为工程名，扩展名为.exe。组建过程中，VC 6.0 会在界面下方的子窗口中显示编译、连接的进度以或相关的错误信息，如图 A-6 所示。

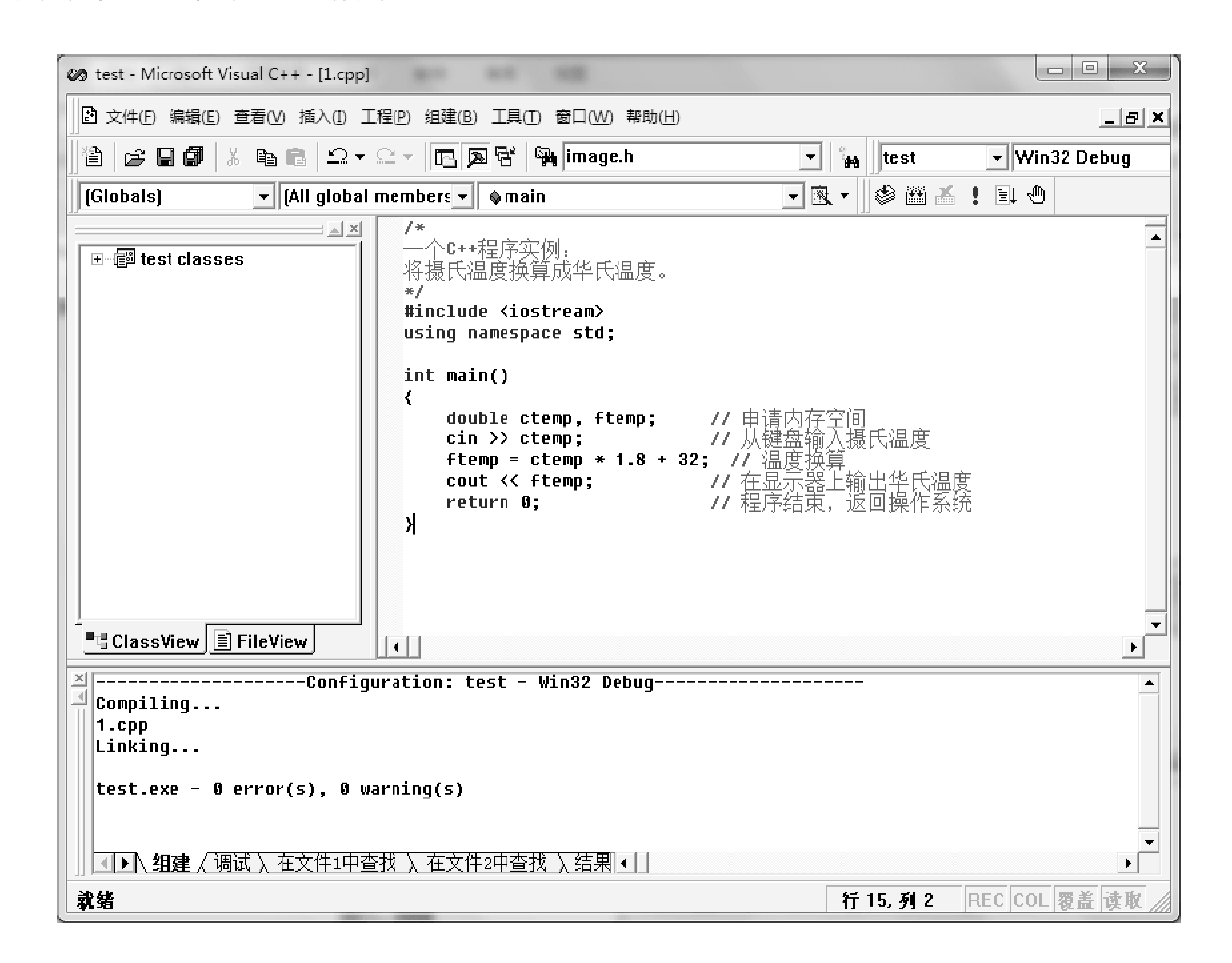



图 A-6 组建过程中的提示信息

VC 6.0 将语法错误分为两类。一类是严重错误(error)，一旦出现严重错误，VC 6.0 将不会生成可执行程序文件；另一类是警示错误(warning)，如果出现警示错误，VC 6.0 将提示错误信息，但仍会生成可执行程序文件。一旦出现错误，程序员应检查原因并修改源程序，然后再重新组建。

初学者编写程序时，组建出现错误是正常的，请不要放弃。优秀程序员都是在不断的出错和改错过程中锤炼成的。

### 4. 执行程序

单击菜单【组建】→【执行】，VC 6.0 将执行上述组建过程所生成的可执行程序文件 test.exe，图 A-7 显示该程序的执行结果。从键盘输入摄氏温度的数值 10，程序将换算成华氏温度，显示换算的结果为 50。程序执行结束后，VC 6.0 将显示提示信息“Press any key to continue”，即按任意键返回 VC 6.0。

执行程序时，程序员应核查执行结果是否正确。如结果不正确，则说明程序算法存

在语义错误(或称为逻辑错误)。程序员应返回源程序，检查并修改错误，然后重新组建、执行。

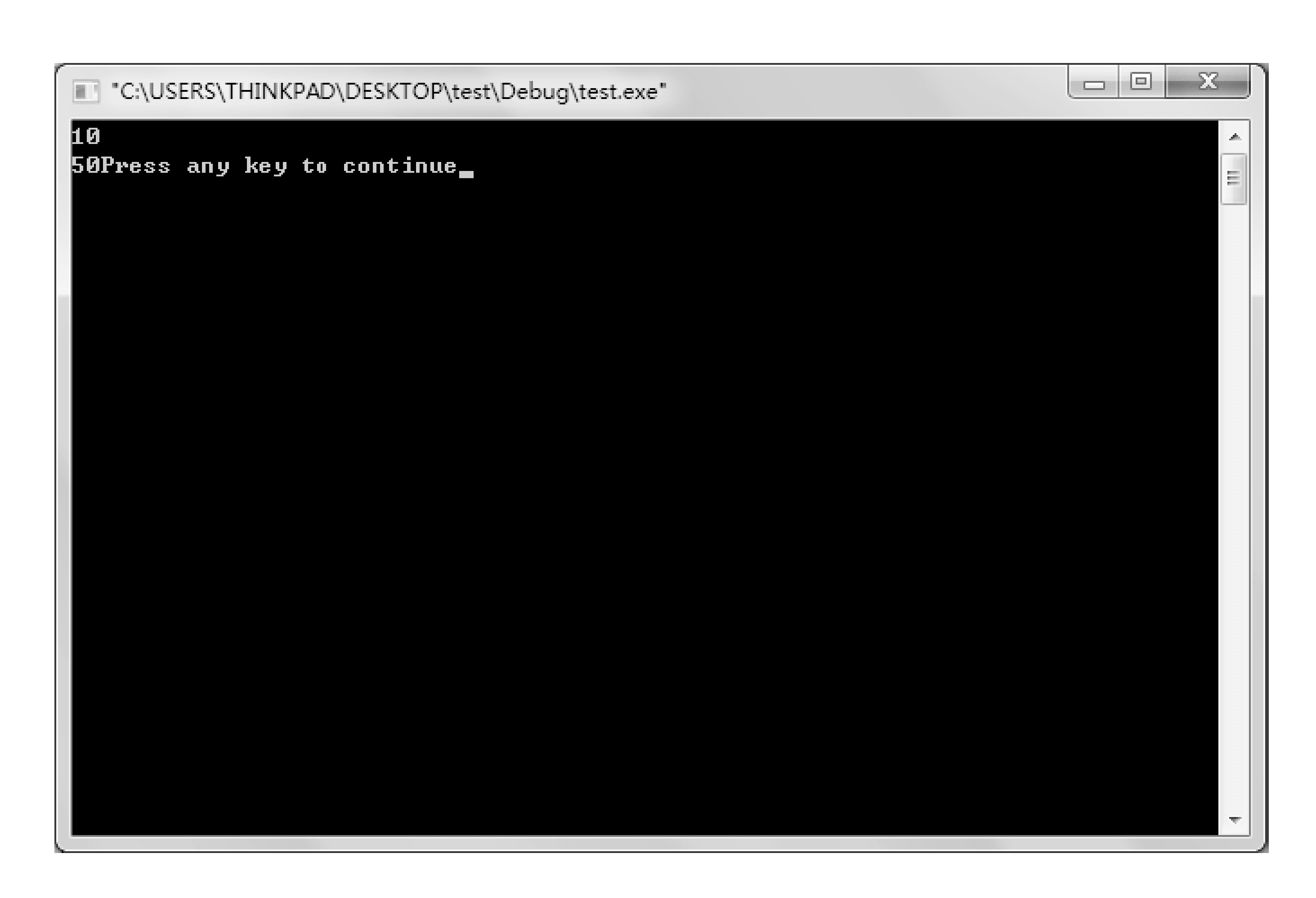



图 A-7　温度换算程序的执行结果

### 5. 重新打开工程

可以重新打开已经创建的工程。以之前创建的工程 test 为例，图 A-8 显示了工程文件夹 test 下的所有内容。

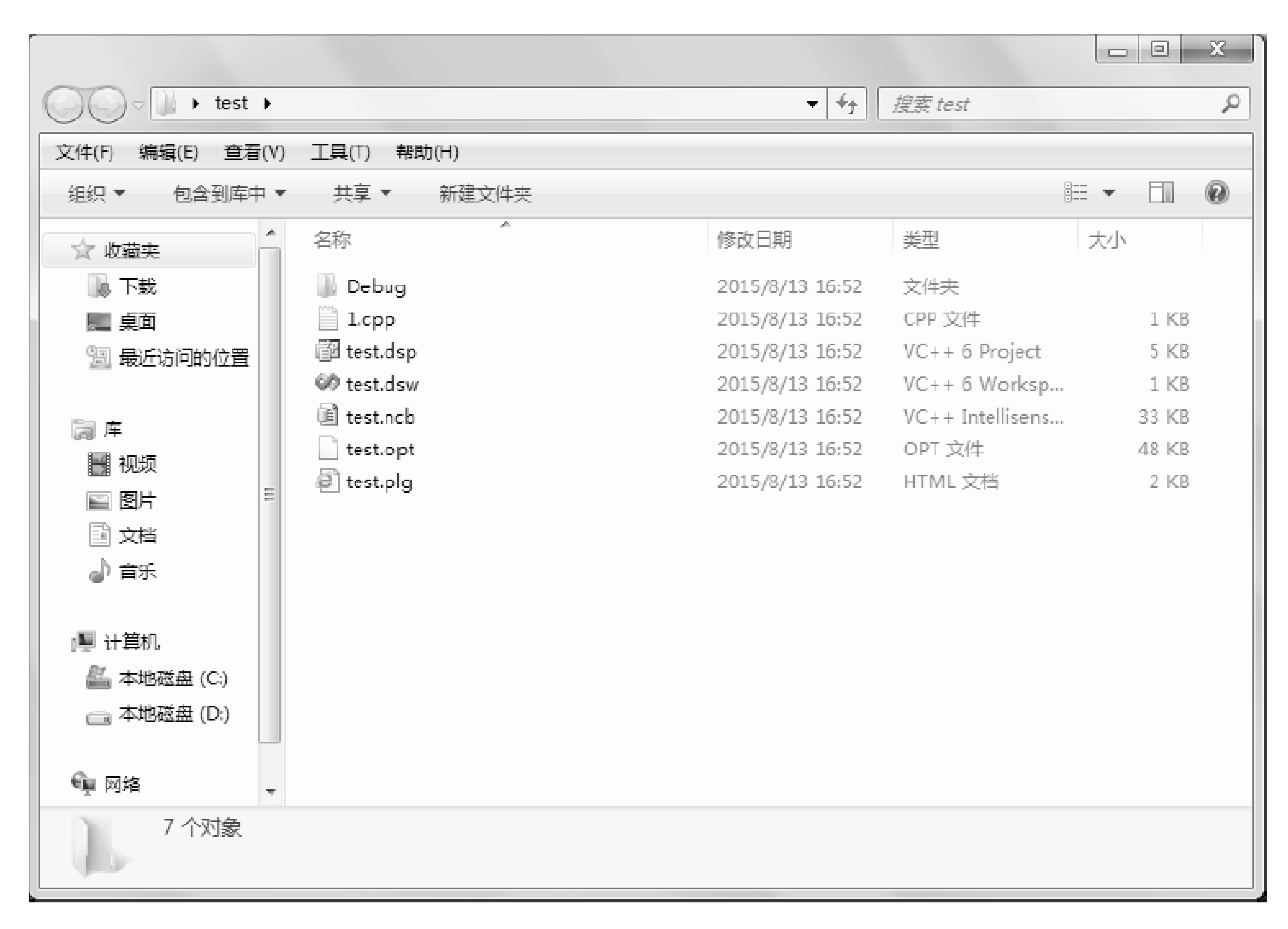



图 A-8　工程 test 的文件夹

在工程 test 的文件夹中，有一个名为 test.dsw 的文件。这个文件是工程的工作空间(developer studio workspace)文件。双击该文件，系统将自动启动 VC 6.0，打开工程 test 并

将其恢复到上次关闭前的状态。通过 VC 6.0 界面左侧的文件视图 FileView 选择打开工程中的源程序文件 Source Files 或头文件 Header Files，如图 A-9 所示。

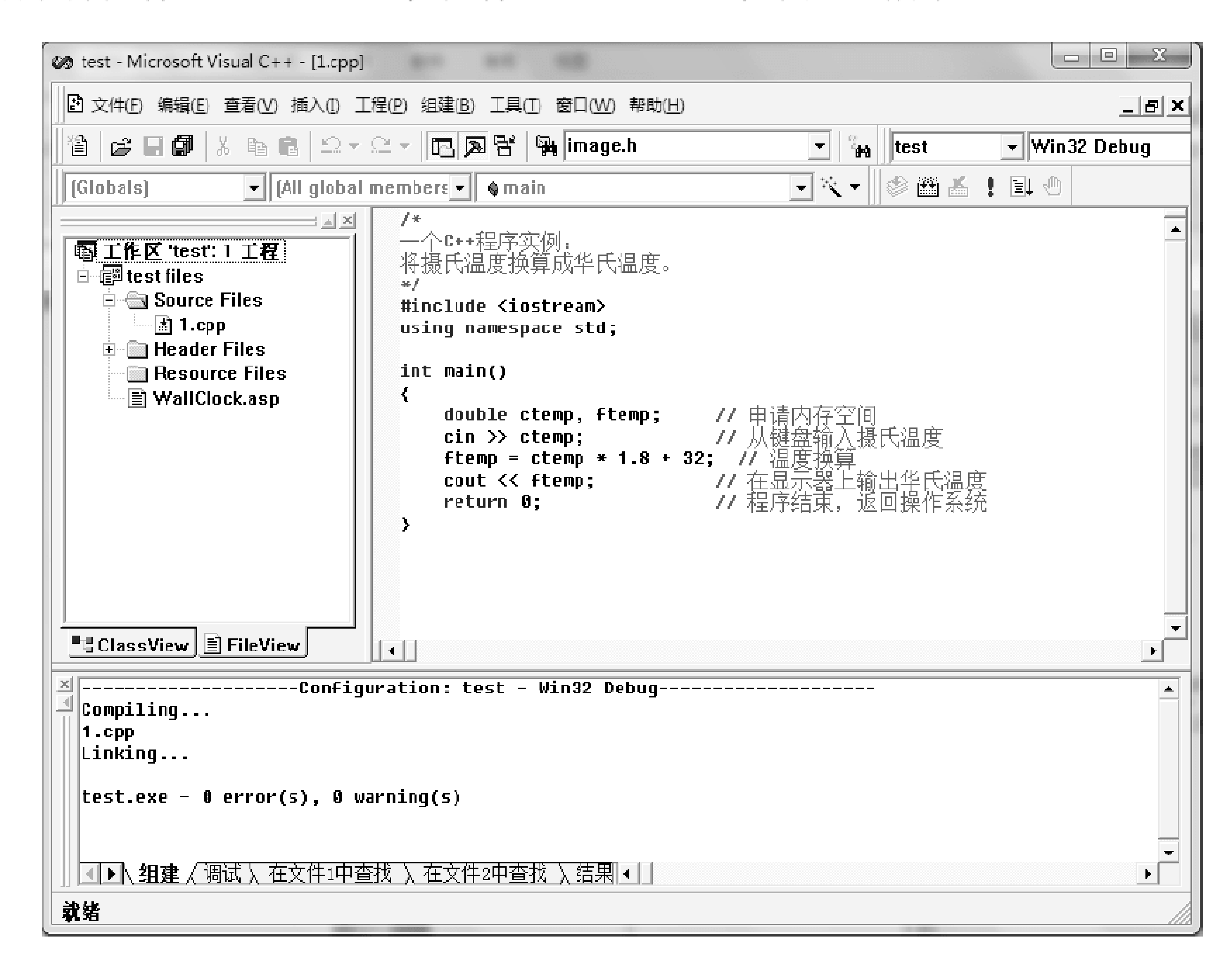



图 A-9　VC 6.0 的文件视图

如果读者使用 Microsoft Visual Studio 2008 或之后的新版集成开发环境，其操作步骤与 VC 6.0 有一些差别。例如新建一个名为 test 的 Win32 控制台应用程序项目(注：VC 6.0 中文版将 project 译成“工程”，而 Visual Studio 中文版将 project 译成“项目”)，则 Visual Studio 系列集成开发环境中的应用程序向导将自动创建一个如下的 C++源程序文件：

```
// test.cpp : 定义控制台应用程序的入口点
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[ ])
{
    return 0;
}
```

该程序有两个主要的特征，一是插入了头文件 stdafx.h，二是将主函数名改为_tmain，并且第二个形参的数据类型是_TCHAR *。之所以做这样的修改，是因为 Visual Studio 系列集成开发环境同时支持 ANSI 编码和 Unicode 编码的程序开发，改用_tmain 可以很方便地在这两种字符编码之间进行切换。初学者可直接将上述源程序文件改写成如下的形式：

```
#include "stdafx.h"                    // 保留该编译预处理指令
#include <iostream>
using namespace std;
int main( )                            // 修改主函数的定义
{
```

```
        // 此处定义函数体代码
        system( "pause" );                    // 该语句的作用是暂停程序执行，以便程序员检查运行结果
        return 0;
    }
```

也可以在新建项目时选择“空项目”，然后自己添加源程序文件，自己输入主函数 main 的定义代码(此时要去掉编译预处理指令：#include "stdafx.h")。

另外，在使用 Visual C++ 6.0 或 Visual Studio 系列集成开发环境编写 Windows 图形用户界面程序(Win32 Application 或 Win32 项目)时，程序员需将主函数名改为 WinMain 或_tWinMain，这是 Windows 图形用户界面程序执行时的起点。

# 附录B 各章"本节习题"参考答案

## 第 1 章　程序设计导论

1.1　计算机硬件结构：D C B C A
1.2　计算机程序：A D B A C
1.3　计算机程序开发：D A D A C
1.4　信息分类与数据类型：C C C C D D D

## 第 2 章　数值计算

2.1　程序中的变量：C D A D A D D D
2.2　程序中的常量：D C D C A D D
2.3　算术运算：D B B C A B
2.4　位运算：B C D D A
2.5　赋值运算：B C D C C
2.6　数据的输入与输出：A C B D B C
2.7　引用与指针：B C C C D B D

## 第 3 章　算法与控制结构

3.1　算法：B D C B A
3.2　布尔类型：A D C B C C C
3.3　选择语句：D A C B D
3.4　循环语句：D D D C C B C D

## 第 4 章　数组与文字处理

4.1　数组：D C B C D D
4.2　指针与数组：A D C A D B
4.3　字符类型：B D D D D
4.4　字符数组与文字处理：C C D D C D

4.5 中文处理：C D D B B

## 第 5 章 结构化程序设计之一

5.1 结构化程序设计方法：D D D D D
5.2 函数的定义和调用：B A B C A B C D D C
5.3 数据的管理策略：A A B B D D
5.4 程序代码和变量的存储原理：B B A D D
5.5 函数间参数传递的三种方式：D B C D A D

## 第 6 章 结构化程序设计之二

6.1 C++源程序的多文件结构：C D D C B
6.2 编译预处理指令：C C D B A
6.3 几种特殊形式的函数：D D D C A B C
6.4 系统函数：C B C C D
6.5 自定义数据类型：C C B D B A

## 第 7 章 面向对象程序设计之一

7.1 面向对象程序设计方法：D D D D D
7.2 面向对象程序的设计过程：B D C D B
7.3 类与对象的语法细则：D B B D D D B C
7.4 对象的构造与析构：D D A A A C C D
7.5 对象的应用：B C C D C
7.6 类中的常成员与静态成员：B B D D C
7.7 类的友元：D C D C

## 第 8 章 面向对象程序设计之二

8.1 重用类代码：C C B D D
8.2 类的组合：A D D D D
8.3 类的继承与派生：D A C D D A A D
8.4 多态性：A C D B
8.5 对象的替换与多态：A B A D C D
8.6 关于多继承的讨论：C B

## 第 9 章 流类库与文件 I/O

9.1 流类库：D B D D
9.2 标准 I/O：D D B B

9.3　文件 I/O：D B A D
9.4　string 类及字符串 I/O：D A A B
9.5　基于 Unicode 编码的流类库：D A

## 第 10 章　C++标准库

10.1　函数模板：D D C B
10.2　类模板：C D A D
10.3　C++标准库：C D
10.4　C++语言的异常处理机制：D C D C B D C D
10.5　数据集合及其处理算法：D D D D D D A D

# 参 考 文 献


[1] 谭浩强. C 语言程序设计教程(第 3 版). 北京：高等教育出版社，2006.

[2] 郑莉. C++语言程序设计(第 3 版). 北京：清华大学出版社，2004.

[3] 郑立华. C++程序设计与应用. 北京：清华大学出版社，2011.

[4] 周幸妮. C 语言程序设计新视角. 西安：西安电子科技大学出版社，2012.

[5] Bernd Bruegge. 面向对象软件工程(第 3 版). 北京：清华大学出版社，2011.

[6] ISO/IEC 14882-2003. Programming languages — C++. ISO, 2003.

[7] ISO/IEC 14882-2011. Information technology — Programming languages — C++. ISO, 2011.

# 图书资源支持

感谢您一直以来对清华版图书的支持和爱护。为了配合本书的使用,本书提供配套的资源,有需求的读者请扫描下方的“书圈”微信公众号二维码,在图书专区下载,也可以拨打电话或发送电子邮件咨询。

如果您在使用本书的过程中遇到了什么问题,或者有相关图书出版计划,也请您发邮件告诉我们,以便我们更好地为您服务。


**我们的联系方式:**

地　　址:北京海淀区双清路学研大厦 A 座 707

邮　　编:100084

电　　话:010－62770175－4604

资源下载:http://www.tup.com.cn

电子邮件:weijj@tup.tsinghua.edu.cn

QQ:883604(请写明您的单位和姓名)


资源下载、样书申请



书圈

**用微信扫一扫右边的二维码,即可关注清华大学出版社公众号“书圈”。**